

Effective Python

<https://github.com/bslatkin>

Published
with GitBook



Table of Contents

Introduction	0
Preface	1
Acknowledgments	2
About the Author	3
Chapter 1: Pythonic Thinking	4
Item 1: Know Which Version of Python You're Using	4.1
Item 2: Follow the PEP8 Style Guide	4.2
Item 3: Know the Differences Between bytes, str, and unicode	4.3
Item 4: Write Helper Functions Instead of Complex Expressions	4.4
Item 5: Know How to Slice Sequences	4.5
Item 6: Avoid Using start, end, and stride in a Single Slice	4.6
Item 7: Use List Comprehensions Instead of map and filter	4.7
Item 8: Avoid More Than Two Expressions in List Comprehensions	4.8
Item 9: Consider Generator Expressions for Large Comprehensions	4.9
Item 10: Prefer enumerate Over range	4.10
Item 11: Use zip to Process Iterators in Parallel	4.11
Item 12: Avoid else Blocks After for and while Loops	4.12
Item 13: Take Advantage of Each Block in try/except/else/finally	4.13
Chapter 2: Functions	5
Item 14: Prefer Exceptions to Returning None	5.1
Item 15: Know How Closures Interact with Variable Scope	5.2
Item 16: Consider Generators Instead of Returning Lists	5.3
Item 17: Be Defensive When Iterating Over Arguments	5.4
Item 18: Reduce Visual Noise with Variable Positional Arguments	5.5
Item 19: Provide Optional Behavior with Keyword Arguments	5.6
Item 20: Use None and Docstrings to Specify Dynamic Default Arguments	5.7
Item 21: Enforce Clarity with Keyword-Only Arguments	5.8
Chapter 3: Classes and Inheritance	6
Item 22: Prefer Helper Classes Over Bookkeeping with Dictionaries and Tuples	
Item 23: Accept Functions for Simple Interfaces Instead of Classes	6.2 6.1

Item 24: Use <code>@classmethod</code> Polymorphism to Construct Objects Generically	6.3
Item 25: Initialize Parent Classes with <code>super</code>	6.4
Item 26: Use Multiple Inheritance Only for Mix-in Utility Classes	6.5
Item 27: Prefer Public Attributes Over Private Ones	6.6
Item 28: Inherit from <code>collections.abc</code> for Custom Container Types	6.7
Chapter 4: Metaclasses and Attributes	7
Item 29: Use Plain Attributes Instead of Get and Set Methods	7.1
Item 30: Consider <code>@property</code> Instead of Refactoring Attributes	7.2
Item 31: Use Descriptors for Reusable <code>@property</code> Methods	7.3
Item 32: Use <code>__getattr__</code> , <code>__getattribute__</code> , and <code>__setattr__</code> for Lazy Attributes	7.4
Item 33: Validate Subclasses with Metaclasses	7.5
Item 34: Register Class Existence with Metaclasses	7.6
Item 35: Annotate Class Attributes with Metaclasses	7.7
Chapter 5: Concurrency and Parallelism	8
Item 36: Use <code>subprocess</code> to Manage Child Processes	8.1
Item 37: Use Threads for Blocking I/O, Avoid for Parallelism	8.2
Item 38: Use Lock to Prevent Data Races in Threads	8.3
Item 39: Use Queue to Coordinate Work Between Threads	8.4
Item 40: Consider Coroutines to Run Many Functions Concurrently	8.5
Item 41: Consider <code>concurrent.futures</code> for True Parallelism	8.6
Chapter 6: Built-in Modules	9
Item 42: Define Function Decorators with <code>functools.wraps</code>	9.1
Item 43: Consider <code>contextlib</code> and <code>with</code> Statements for Reusable <code>try/finally</code> Behavior	
Item 44: Make pickle Reliable with <code>copyreg</code>	9.3 9.2
Item 45: Use <code>datetime</code> Instead of <code>time</code> for Local Clocks	9.4
Item 46: Use Built-in Algorithms and Data Structures	9.5
Item 47: Use <code>decimal</code> When Precision Is Paramount	9.6
Item 48: Know Where to Find Community-Built Modules	9.7
Chapter 7: Collaboration	10
Item 49: Write Docstrings for Every Function, Class, and Module	10.1
Item 50: Use Packages to Organize Modules and Provide Stable APIs	10.2
Item 51: Define a Root Exception to Insulate Callers from APIs	10.3
Item 52: Know How to Break Circular Dependencies	10.4
Item 53: Use Virtual Environments for Isolated and Reproducible Dependencies	

Chapter 8: Production	11	10.5
Item 54: Consider Module-Scoped Code to Configure Deployment Environments		
Item 55: Use repr Strings for Debugging Output	11.2	11.1
Item 56: Test Everything with unittest		11.3
Item 57: Consider Interactive Debugging with pdb		11.4
Item 58: Profile Before Optimizing		11.5
Item 59: Use tracemalloc to Understand Memory Usage and Leaks Index		11.6

Introduction

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 PEP8 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.

Pythonic Thinking

The idioms of a programming language are defined by its users. Over the years, the Python community has come to use the adjective Pythonic to describe code that follows a particular style. The Pythonic style isn't regimented or enforced by the compiler. It has emerged over time through experience using the language and working with others. Python programmers prefer to be explicit, to choose simple over complex, and to maximize readability (type import this).

Programmers familiar with other languages may try to write Python as if it's C++, Java, or whatever they know best. New programmers may still be getting comfortable with the vast range of concepts expressible in Python. It's important for everyone to know the best—the Pythonic—way to do the most common things in Python. These patterns will affect every program you write.

Item 1: Know Which Version of Python You're Using

Throughout this book, the majority of example code is in the syntax of Python 3.4 (released March 17, 2014). This book also provides some examples in the syntax of Python 2.7 (released July 3, 2010) to highlight important differences. Most of my advice applies to all of the popular Python runtimes: CPython, Jython, IronPython, PyPy, etc.

Many computers come with multiple versions of the standard CPython runtime preinstalled. However, the default meaning of `python` on the command-line may not be clear. `python` is usually an alias for `python2.7`, but it can sometimes be an alias for older versions like `python2.6` or `python2.5`. To find out exactly which version of Python you're using, you can use the `--version` flag.

```
$ python --version
Python 2.7.8
```

Python 3 is usually available under the name `python3`.

```
$ python3 --version
Python 3.4.2
```

You can also figure out the version of Python you're using at runtime by inspecting values in the `sys` built-in module.

```
import sys
print(sys.version_info)
print(sys.version)

>>>

sys.version_info(major=3, minor=4, micro=2, releaselevel='final', serial=0) 3.4.2 (default
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.51])
```

Python 2 and Python 3 are both actively maintained by the Python community. Development on Python 2 is frozen beyond bug fixes, security improvements, and backports to ease the transition from Python 2 to Python 3. Helpful tools like the `2to3` and `six` exist to make it easier to adopt Python 3 going forward.

Python 3 is constantly getting new features and improvements that will never be added to Python 2. As of the writing of this book, the majority of Python's most common open source libraries are compatible with Python 3. I strongly encourage you to use Python 3 for your next Python project.

Things to Remember

- There are two major versions of Python still in active use: Python 2 and Python 3.
- There are multiple popular runtimes for Python: CPython, Jython, IronPython, PyPy, etc.
- Be sure that the command-line for running Python on your system is the version you expect it to be.
- Prefer Python 3 for your next project because that is the primary focus of the Python community.

Item 2: Follow the PEP8 Style Guide

Python Enhancement Proposal #8, otherwise known as PEP8, is the style guide for how to format Python code. You are welcome to write Python code however you want, as long as it has valid syntax. However, using a consistent style makes your code more approachable and easier to read. Sharing a common style with other Python programmers in the larger community facilitates collaboration on projects. But even if you are the only one who will ever read your code, following the style guide will make it easier to change things later.

PEP8 has a wealth of details about how to write clear Python code. It continues to be updated as the Python language evolves. It's worth reading the whole guide online (<http://www.python.org/dev/peps/pep-0008/>). Here are a few rules you should be sure to follow:

Whitespace: In Python, whitespace is syntactically significant. Python programmers are especially sensitive to the effects of whitespace on code clarity.

- Use spaces instead of tabs for indentation.
- Use four spaces for each level of syntactically significant indenting.
- Lines should be 79 characters in length or less.
- Continuations of long expressions onto additional lines should be indented by four extra spaces from their normal indentation level.
- In a file, functions and classes should be separated by two blank lines.
- In a class, methods should be separated by one blank line.
- Don't put spaces around list indexes, function calls, or keyword argument assignments.
- Put one—and only one—space before and after variable assignments.

Naming: PEP8 suggests unique styles of naming for different parts in the language. This makes it easy to distinguish which type corresponds to each name when reading code.

- Functions, variables, and attributes should be in lowercase_underscore format.
- Protected instance attributes should be in _leading_underscore format.
- Private instance attributes should be in __double_leading_underscore format.
- Classes and exceptions should be in CapitalizedWord format.
- Module-level constants should be in ALL_CAPS format.

- Instance methods in classes should use `self` as the name of the first parameter (which refers to the object).
- Class methods should use `cls` as the name of the first parameter (which refers to the class).

Expressions and Statements: The Zen of Python states: “There should be one—and preferably only one—obvious way to do it.” PEP8 attempts to codify this style in its guidance for expressions and statements.

- Use inline negation (if `a` is not `b`) instead of negation of positive expressions (if not `a` is `b`).
- Don’t check for empty values (like `[]` or `''`) by checking the length (if `len(somelist) == 0`). Use `if not somelist` and assume empty values implicitly evaluate to `False`.
- The same thing goes for non-empty values (like `[1]` or `'hi'`). The statement `if somelist` is implicitly `True` for non-empty values.
- Avoid single-line `if` statements, `for` and `while` loops, and `except` compound statements. Spread these over multiple lines for clarity.
- Always put `import` statements at the top of a file.
- Always use absolute names for modules when importing them, not names relative to the current module’s own path. For example, to import the `foo` module from the `bar` package, you should do `from bar import foo`, not just `import foo`.
- If you must do relative imports, use the explicit syntax `from . import foo`.
- Imports should be in sections in the following order: standard library modules, third-party modules, your own modules. Each subsection should have imports in alphabetical order.

Note

The Pylint tool (<http://www.pylint.org/>) is a popular static analyzer for Python source code. Pylint provides automated enforcement of the PEP8 style guide and detects many other types of common errors in Python programs.

Things to Remember

- Always follow the PEP8 style guide when writing Python code.
- Sharing a common style with the larger Python community facilitates collaboration with others.

- Using a consistent style makes it easier to modify your own code later.

Item 3: Know the Differences Between bytes, str, and unicode

In Python 3, there are two types that represent sequences of characters: bytes and str. Instances of bytes contain raw 8-bit values. Instances of str contain Unicode characters.

In Python 2, there are two types that represent sequences of characters: str and unicode. In contrast to Python 3, instances of str contain raw 8-bit values. Instances of unicode contain Unicode characters.

There are many ways to represent Unicode characters as binary data (raw 8-bit values). The most common encoding is UTF-8. Importantly, str instances in Python 3 and unicode instances in Python 2 do not have an associated binary encoding. To convert Unicode characters to binary data, you must use the encode method. To convert binary data to Unicode characters, you must use the decode method.

When you're writing Python programs, it's important to do encoding and decoding of Unicode at the furthest boundary of your interfaces. The core of your program should use Unicode character types (str in Python 3, unicode in Python 2) and should not assume anything about character encodings. This approach allows you to be very accepting of alternative text encodings (such as Latin-1, Shift JIS, and Big5) while being strict about your output text encoding (ideally, UTF-8).

The split between character types leads to two common situations in Python code:

- You want to operate on raw 8-bit values that are UTF-8-encoded characters (or some other encoding).
- You want to operate on Unicode characters that have no specific encoding.

You'll often need two helper functions to convert between these two cases and to ensure that the type of input values matches your code's expectations.

In Python 3, you'll need one method that takes a str or bytes and always returns a str.

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value    # Instance of str
```

You'll need another method that takes a str or bytes and always returns a bytes.

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value    # Instance of bytes
```

In Python 2, you'll need one method that takes a str or unicode and always returns a unicode.

```
# Python 2

def to_unicode(unicode_or_str):
    if isinstance(unicode_or_str, str):
        value = unicode_or_str.decode('utf-8')
    else:
        value = unicode_or_str
    return value    # Instance of unicode
```

You'll need another method that takes str or unicode and always returns a str.

```
# Python 2

def to_str(unicode_or_str):
    if isinstance(unicode_or_str, unicode):
        value = unicode_or_str.encode('utf-8')
    else:
        value = unicode_or_str
    return value    # Instance of str
```

There are two big gotchas when dealing with raw 8-bit values and Unicode characters in Python.

The first issue is that in Python 2, unicode and str instances seem to be the same type when a str only contains 7-bit ASCII characters.

- You can combine such a str and unicode together using the + operator.
- You can compare such str and unicode instances using equality and inequality operators.
- You can use unicode instances for format strings like '%s'.

All of this behavior means that you can often pass a str or unicode instance to a function expecting one or the other and things will just work (as long as you're only dealing with 7-bit ASCII). In Python 3, bytes and str instances are never equivalent— not even the empty

string—so you must be more deliberate about the types of character sequences that you're passing around.

The second issue is that in Python 3, operations involving file handles (returned by the open built-in function) default to UTF-8 encoding. In Python 2, file operations default to binary encoding. This causes surprising failures, especially for programmers accustomed to Python 2.

For example, say you want to write some random binary data to a file. In Python 2, this works. In Python 3, this breaks.

```
with open('/tmp/random.bin', 'w') as f:
    f.write(os.urandom(10))

>>>

TypeError: must be str, not bytes
```

The cause of this exception is the new encoding argument for open that was added in Python 3. This parameter defaults to 'utf-8'. That makes read and write operations on file handles expect str instances containing Unicode characters instead of bytes instances containing binary data.

To make this work properly, you must indicate that the data is being opened in write binary mode ('wb') instead of write character mode ('w'). Here, I use open in a way that works correctly in Python 2 and Python 3:

```
with open('/tmp/random.bin', 'wb') as f:
    f.write(os.urandom(10))
```

This problem also exists for reading data from files. The solution is the same: Indicate binary mode by using 'rb' instead of 'r' when opening a file.

Things to Remember

- In Python 3, bytes contains sequences of 8-bit values, str contains sequences of Unicode characters. bytes and str instances can't be used together with operators (like > or +).
- In Python 2, str contains sequences of 8-bit values, unicode contains sequences of Unicode characters. str and unicode can be used together with operators if the str only contains 7-bit ASCII characters.

- Use helper functions to ensure that the inputs you operate on are the type of character sequence you expect (8-bit values, UTF-8 encoded characters, Unicode characters, etc.).
- If you want to read or write binary data to/from a file, always open the file using a binary mode (like 'rb' or 'wb').

Item 4: Write Helper Functions Instead of Complex Expressions

Python's pithy syntax makes it easy to write single-line expressions that implement a lot of logic. For example, say you want to decode the query string from a URL. Here, each query string parameter represents an integer value:

```
from urllib.parse import parse_qs

my_values = parse_qs('red=5&blue=0&green=', keep_blank_values=True)
print(repr(my_values))

>>>

{'red': ['5'], 'green': [''], 'blue': ['0']}
```

Some query string parameters may have multiple values, some may have single values, some may be present but have blank values, and some may be missing entirely. Using the `get` method on the result dictionary will return different values in each circumstance.

```
print('Red:      ', my_values.get('red'))
print('Green:    ', my_values.get('green'))
print('Opacity:  ', my_values.get('opacity'))

>>>

Red:          ['5']
Green:        ['']
Opacity:      None
```

It'd be nice if a default value of 0 was assigned when a parameter isn't supplied or is blank. You might choose to do this with Boolean expressions because it feels like this logic doesn't merit a whole `if` statement or helper function quite yet.

Python's syntax makes this choice all too easy. The trick here is that the empty string, the empty list, and zero all evaluate to `False` implicitly. Thus, the expressions below will evaluate to the subexpression after the `or` operator when the first subexpression is `False`.

```
# For query string 'red=5&blue=0&green='

red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0

print('Red:      %r' % red)
print('Green:    %r' % green)
print('Opacity:  %r' % opacity)

>>>

Red:      '5'
Green:    0
Opacity:  0
```

The red case works because the key is present in the `my_values` dictionary. The value is a list with one member: the string '5'. This string implicitly evaluates to True, so red is assigned to the first part of the or expression.

The green case works because the value in the `my_values` dictionary is a list with one member: an empty string. The empty string implicitly evaluates to False, causing the or expression to evaluate to 0.

The opacity case works because the value in the `my_values` dictionary is missing altogether. The behavior of the `get` method is to return its second argument if the key doesn't exist in the dictionary. The default value in this case is a list with one member, an empty string. When opacity isn't found in the dictionary, this code does exactly the same thing as the green case.

However, this expression is difficult to read and it still doesn't do everything you need. You'd also want to ensure that all the parameter values are integers so you can use them in mathematical expressions. To do that, you'd wrap each expression with the `int` built-in function to parse the string as an integer.

```
red = int(my_values.get('red', [''])[0] or 0)
```

This is now extremely hard to read. There's so much visual noise. The code isn't approachable. A new reader of the code would have to spend too much time picking apart the expression to figure out what it actually does. Even though it's nice to keep things short, it's not worth trying to fit this all on one line.

Python 2.5 added if/else conditional—or ternary—expressions to make cases like this clearer while keeping the code short.


```
red = my_values.get('red', [])  
red = int(red[0]) if red[0] else 0
```

This is better. For less complicated situations, if/else conditional expressions can make things very clear. But the example above is still not as clear as the alternative of a full if/else statement over multiple lines. Seeing all of the logic spread out like this makes the dense version seem even more complex.

```
green = my_values.get('green', [])  
if green[0]:  
    green = int(green[0])  
else:  
    green = 0
```

Writing a helper function is the way to go, especially if you need to use this logic repeatedly.

```
def get_first_int(values, key, default=0):  
    found = values.get(key, [])  
    if found[0]:  
        found = int(found[0])  
    else:  
        found = default  
    return found
```

The calling code is much clearer than the complex expression using or and the two-line version using the if/else expression.

```
green = get_first_int(my_values, 'green')
```

As soon as your expressions get complicated, it's time to consider splitting them into smaller pieces and moving logic into helper functions. What you gain in readability always outweighs what brevity may have afforded you. Don't let Python's pithy syntax for complex expressions get you into a mess like this.

Things to Remember

- Python's syntax makes it all too easy to write single-line expressions that are overly complicated and difficult to read.
- Move complex expressions into helper functions, especially if you need to use the same logic repeatedly.
- The if/else expression provides a more readable alternative to using Boolean operators like or and and in expressions.

Item 5: Know How to Slice Sequences

Python includes syntax for slicing sequences into pieces. Slicing lets you access a subset of a sequence's items with minimal effort. The simplest uses for slicing are the built-in types `list`, `str`, and `bytes`. Slicing can be extended to any Python class that implements the `__getitem__` and `__setitem__` special methods (see Item 28: "Inherit from `collections.abc` for Custom Container Types").

The basic form of the slicing syntax is `somelist[start:end]`, where `start` is inclusive and `end` is exclusive.

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

print('First four:', a[:4])
print('Last four: ', a[-4:])
print('Middle two:', a[3:-3])

>>>

First four: ['a', 'b', 'c', 'd']
Last four:  ['e', 'f', 'g', 'h']
Middle two: ['d', 'e']
```

When slicing from the start of a list, you should leave out the zero index to reduce visual noise.

```
assert a[:5] == a[0:5]
```

When slicing to the end of a list, you should leave out the final index because it's redundant.

```
assert a[5:] == a[5:len(a)]
```

Using negative numbers for slicing is helpful for doing offsets relative to the end of a list. All of these forms of slicing would be clear to a new reader of your code. There are no surprises, and I encourage you to use these variations.

```

a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     # ['e', 'f', 'g', 'h']
a[-3:]    # ['f', 'g', 'h']
a[2:5]    # ['c', 'd', 'e']
a[2:-1]   # ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  # ['f', 'g']

```

Slicing deals properly with start and end indexes that are beyond the boundaries of the list. That makes it easy for your code to establish a maximum length to consider for an input sequence.

```
first_twenty_items = a[:20] last_twenty_items = a[-20:]
```

In contrast, accessing the same index directly causes an exception.

```

a[20]

>>>

IndexError: list index out of range

```

Note

Beware that indexing a list by a negative variable is one of the few situations in which you can get surprising results from slicing. For example, the expression `somelist[-n:]` will work fine when `n` is greater than one (e.g., `somelist[-3:]`). However, when `n` is zero, the expression `somelist[-0:]` will result in a copy of the original list.

The result of slicing a list is a whole new list. References to the objects from the original list are maintained. Modifying the result of slicing won't affect the original list.

```

b = a[4:]
print('Before: ', b)
b[1] = 99
print('After: ', b)
print('No change:', a)

>>>
Before: ['e', 'f', 'g', 'h']
After : ['e', 99, 'g', 'h']
No change: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

```

When used in assignments, slices will replace the specified range in the original list. Unlike tuple assignments (like `a, b = c[:2]`), the length of slice assignments don't need to be the same. The values before and after the assigned slice will be preserved. The list will grow or shrink to accommodate the new values.

```
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After   ', a)
>>>
Before      ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
After       ['a', 'b', 99, 22, 14, 'h']
```

If you leave out both the start and the end indexes when slicing, you'll end up with a copy of the original list.

```
b = a[:]
assert b == a and b is not a
```

If you assign a slice with no start or end indexes, you'll replace its entire contents with a copy of what's referenced (instead of allocating a new list).

```
b = a
print('Before', a)
a[:] = [101, 102, 103]
assert a is b      # Still the same list object
print('After ', a)  # Now has different contents

>>>

Before ['a', 'b', 99, 22, 14, 'h'] After    [101, 102, 103]
```

Things to Remember

- Avoid being verbose: Don't supply 0 for the start index or the length of the sequence for the end index.
- Slicing is forgiving of start or end indexes that are out of bounds, making it easy to express slices on the front or back boundaries of a sequence (like `a[:20]` or `a[-20:]`).
- Assigning to a list slice will replace that range in the original sequence with what's referenced even if their lengths are different.

Item 6: Avoid Using start, end, and stride in a Single Slice

In addition to basic slicing (see Item 5: “Know How to Slice Sequences”), Python has special syntax for the stride of a slice in the form `somelist [start:end:stride]`. This lets you take every *n*th item when slicing a sequence. For example, the stride makes it easy to group by even and odd indexes in a list.

```
a = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = a[::2]
evens = a[1::2]
print(odds)
print(evens)

>>>

['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

The problem is that the stride syntax often causes unexpected behavior that can introduce bugs. For example, a common Python trick for reversing a byte string is to slice the string with a stride of -1.

```
x = b'mongoose'
y = x[::-1]
print(y)

>>>
b'esoongnom'
```

That works well for byte strings and ASCII characters, but it will break for Unicode characters encoded as UTF-8 byte strings.

```
w = '谢谢'
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')

>>>

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd9 in position 0: invalid start byte
```

Are negative strides besides -1 useful? Consider the following examples.

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[::2] # ['a', 'c', 'e', 'g']
a[::-2] # ['h', 'f', 'd', 'b']
```

Here, ::2 means select every second item starting at the beginning. Trickier, ::-2 means select every second item starting at the end and moving backwards.

What do you think 2::2 means? What about -2::-2 vs. -2:2:-2 vs. 2:2:-2?

```
a[2::2] # ['c', 'e', 'g']
a[-2::-2] # ['g', 'e', 'c', 'a']
a[-2:2:-2] # ['g', 'e']
a[2:2:-2] # []
```

The point is that the stride part of the slicing syntax can be extremely confusing. Having three numbers within the brackets is hard enough to read because of its density. Then it's not obvious when the start and end indexes come into effect relative to the stride value, especially when stride is negative.

To prevent problems, avoid using stride along with start and end indexes. If you must use a stride, prefer making it a positive value and omit start and end indexes. If you must use stride with start or end indexes, consider using one assignment to stride and another to slice.

```
b = a[::2] # ['a', 'c', 'e', 'g']
c = b[1:-1] # ['c', 'e']
```

Slicing and then striding will create an extra shallow copy of the data. The first operation should try to reduce the size of the resulting slice by as much as possible. If your program can't afford the time or memory required for two steps, consider using the `itertools` built-in module's `islice` method (see Item 46: "Use Built-in Algorithms and Data Structures"), which doesn't permit negative values for start, end, or stride.

Things to Remember

- Specifying start, end, and stride in a slice can be extremely confusing.
- Prefer using positive stride values in slices without start or end indexes. Avoid negative stride values if possible.

- Avoid using start, end, and stride together in a single slice. If you need all three parameters, consider doing two assignments (one to slice, another to stride) or using `islice` from the `itertools` built-in module.

Item 7: Use List Comprehensions Instead of map and filter

Python provides compact syntax for deriving one list from another. These expressions are called list comprehensions. For example, say you want to compute the square of each number in a list. You can do this by providing the expression for your computation and the input sequence to loop over.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = [x**2 for x in a]
print(squares)

>>>

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unless you're applying a single-argument function, list comprehensions are clearer than the map built-in function for simple cases. map requires creating a lambda function for the computation, which is visually noisy.

```
squares = map(lambda x: x ** 2, a)
```

Unlike map, list comprehensions let you easily filter items from the input list, removing corresponding outputs from the result. For example, say you only want to compute the squares of the numbers that are divisible by 2. Here, I do this by adding a conditional expression to the list comprehension after the loop:

```
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)

>>>

[4, 16, 36, 64, 100]
```

The filter built-in function can be used along with map to achieve the same outcome, but it is much harder to read.

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

Dictionaries and sets have their own equivalents of list comprehensions. These make it easy to create derivative data structures when writing algorithms.

```
chile_ranks = {'ghost': 1, 'habanero': 2, 'cayenne': 3}
rank_dict = {rank: name for name, rank in chile_ranks.items()}
chile_len_set = {len(name) for name in rank_dict.values()}
print(rank_dict)
print(chile_len_set)
```

```
>>>
```

```
{1: 'ghost', 2: 'habanero', 3: 'cayenne'}
{8, 5, 7}
```

Things to Remember

- List comprehensions are clearer than the map and filter built-in functions because they don't require extra lambda expressions.
- List comprehensions allow you to easily skip items from the input list, a behavior map doesn't support without help from filter.
- Dictionaries and sets also support comprehension expressions.

Item 8: Avoid More Than Two Expressions in List Comprehensions

Beyond basic usage (see Item 7: “Use List Comprehensions Instead of map and filter”), list comprehensions also support multiple levels of looping. For example, say you want to simplify a matrix (a list containing other lists) into one flat list of all cells. Here, I do this with a list comprehension by including two for expressions. These expressions run in the order provided from left to right.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)

>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The example above is simple, readable, and a reasonable usage of multiple loops. Another reasonable usage of multiple loops is replicating the two-level deep layout of the input list. For example, say you want to square the value in each cell of a two-dimensional matrix. This expression is noisier because of the extra `[]` characters, but it’s still easy to read.

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)

>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

If this expression included another loop, the list comprehension would get so long that you’d have to split it over multiple lines.

```
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    # ...
]

flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

At this point, the multiline comprehension isn't much shorter than the alternative. Here, I produce the same result using normal loop statements. The indentation of this version makes the looping clearer than the list comprehension.

```
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

List comprehensions also support multiple if conditions. Multiple conditions at the same loop level are an implicit and expression. For example, say you want to filter a list of numbers to only even values greater than four. These two list comprehensions are equivalent.

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

Conditions can be specified at each level of looping after the for expression. For example, say you want to filter a matrix so the only cells remaining are those divisible by 3 in rows that sum to 10 or higher. Expressing this with list comprehensions is short, but extremely difficult to read.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)

>>>

[[6], [9]]
```

Though this example is a bit convoluted, in practice you'll see situations arise where such expressions seem like a good fit. I strongly encourage you to avoid using list comprehensions that look like this. The resulting code is very difficult for others to comprehend. What you save in the number of lines doesn't outweigh the difficulties it could cause later.

The rule of thumb is to avoid using more than two expressions in a list comprehension. This could be two conditions, two loops, or one condition and one loop. As soon as it gets more complicated than that, you should use normal if and for statements and write a helper function (see Item 16: "Consider Generators Instead of Returning Lists").

Things to Remember

- List comprehensions support multiple levels of loops and multiple conditions per loop level.
- List comprehensions with more than two expressions are very difficult to read and should be avoided.

Item 25: Initialize Parent Classes with `super`

The old way to initialize a parent class from a child class is to directly call the parent class's `__init__` method with the child instance.

```
class MyBaseClass(object):

    def __init__(self, value): self.value = value

class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
```

This approach works fine for simple hierarchies but breaks down in many cases.

If your class is affected by multiple inheritance (something to avoid in general; see Item 26: “Use Multiple Inheritance Only for Mix-in Utility Classes”), calling the superclasses' `__init__` methods directly can lead to unpredictable behavior.

One problem is that the `__init__` call order isn't specified across all subclasses. For example, here I define two parent classes that operate on the instance's value field:

```
class TimesTwo(object):
    def __init__(self):
        self.value *= 2

class PlusFive(object):
    def __init__(self):
        self.value += 5
```

This class defines its parent classes in one ordering.

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

And constructing it produces a result that matches the parent class ordering.

```
foo = OneWay(5)

print('First ordering is (5 * 2) + 5 =', foo.value)

>>>

First ordering is (5 * 2) + 5 = 15
```

Here's another class that defines the same parent classes but in a different ordering:

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):

    def __init__(self, value):
        MyBaseClass.__init__(self, value) TimesTwo.__init__(self) PlusFive.__init__(self)
```

However, I left the calls to the parent class constructors `PlusFive.init` and `TimesTwo.init` in the same order as before, causing this class's behavior not to match the order of the parent classes in its definition.

```
bar = AnotherWay(5)

print('Second ordering still is', bar.value)

>>>

Second ordering still is 15
```

Another problem occurs with diamond inheritance. Diamond inheritance happens when a subclass inherits from two separate classes that have the same superclass somewhere in the hierarchy. Diamond inheritance causes the common superclass's `init` method to run multiple times, causing unexpected behavior. For example, here I define two child classes that inherit from `MyBaseClass`.

```
class TimesFive(MyBaseClass): def __init__(self, value):
    MyBaseClass.__init__(self, value) self.value *= 5

class PlusTwo(MyBaseClass): def __init__(self, value):
    MyBaseClass.__init__(self, value) self.value += 2
```

Then, I define a child class that inherits from both of these classes, making `MyBaseClass` the top of the diamond.

```
class ThisWay(TimesFive, PlusTwo): def __init__(self, value):
    TimesFive.__init__(self, value) PlusTwo.__init__(self, value)

foo = ThisWay(5)

print('Should be (5 * 5) + 2 = 27 but is', foo.value)

>>>

Should be (5 * 5) + 2 = 27 but is 7
```

The output should be 27 because $(5 * 5) + 2 = 27$. But the call to the second parent class's constructor, `PlusTwo.init`, causes `self.value` to be reset back to 5 when `MyBaseClass.init` gets called a second time.

To solve these problems, Python 2.2 added the `super` built-in function and defined the method resolution order (MRO). The MRO standardizes which superclasses are initialized before others (e.g., depth-first, left-to-right). It also ensures that common superclasses in diamond hierarchies are only run once.

Here, I create a diamond-shaped class hierarchy again, but this time I use `super` (in the Python 2 style) to initialize the parent class:

```
# Python 2

class TimesFiveCorrect(MyBaseClass): def __init__(self, value):
    super(TimesFiveCorrect, self).__init__(value) self.value *= 5

class PlusTwoCorrect(MyBaseClass): def __init__(self, value):
    super(PlusTwoCorrect, self).__init__(value) self.value += 2
```

Now the top part of the diamond, `MyBaseClass.init`, is only run a single time. The other parent classes are run in the order specified in the class statement.

```
# Python 2

class GoodWay(TimesFiveCorrect, PlusTwoCorrect): def __init__(self, value):
    super(GoodWay, self).__init__(value)

foo = GoodWay(5)

print 'Should be 5 * (5 + 2) = 35 and is', foo.value

>>>

Should be 5 * (5 + 2) = 35 and is 35
```

This order may seem backwards at first. Shouldn't `TimesFiveCorrect.init` have run first? Shouldn't the result be $(5 * 5) + 2 = 27$? The answer is no. This ordering matches what the MRO defines for this class. The MRO ordering is available on a class method called `mro`.


```

from pprint import pprint pprint(GoodWay.mro())

>>>

[<class '__main__.GoodWay'>,
 <class '__main__.TimesFiveCorrect'>, <class '__main__.PlusTwoCorrect'>, <class '__main__.
 <class 'object'>]

```

When I call `GoodWay(5)`, it in turn calls `TimesFiveCorrect.init`, which calls `PlusTwoCorrect.init`, which calls `MyBaseClass.init`. Once this reaches the top of the diamond, then all of the initialization methods actually do their work in the opposite order from how their `init` functions were called. `MyBaseClass.init` assigns the value to 5. `PlusTwoCorrect.init` adds 2 to make value equal 7. `TimesFiveCorrect.init` multiplies it by 5 to make value equal 35.

The super built-in function works well, but it still has two noticeable problems in Python 2:

Its syntax is a bit verbose. You have to specify the class you're in, the self object, the method name (usually `init`), and all the arguments. This construction can be confusing to new Python programmers.

You have to specify the current class by name in the call to `super`. If you ever change the class's name—a very common activity when improving a class hierarchy—you also need to update every call to `super`.

Thankfully, Python 3 fixes these issues by making calls to `super` with no arguments equivalent to calling `super` with `class` and `self` specified. In Python 3, you should always use `super` because it's clear, concise, and always does the right thing.

```

class Explicit(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value * 2)

class Implicit(MyBaseClass):
    def __init__(self, value):
        super().__init__(value * 2)

assert Explicit(10).value == Implicit(10).value

```

This works because Python 3 lets you reliably reference the current class in methods using the `class` variable. This doesn't work in Python 2 because `class` isn't defined. You may guess that you could use `self.class` as an argument to `super`, but this breaks because of the

way `super` is implemented in Python 2.

Things to Remember

Python's standard method resolution order (MRO) solves the problems of superclass initialization order and diamond inheritance.

Always use the `super` built-in function to initialize parent classes.

Item 26: Use Multiple Inheritance Only for Mix-in Utility Classes

Python is an object-oriented language with built-in facilities for making multiple inheritance tractable (see Item 25: “Initialize Parent Classes with `super`”). However, it’s better to avoid multiple inheritance altogether.

If you find yourself desiring the convenience and encapsulation that comes with multiple inheritance, consider writing a mix-in instead. A mix-in is a small class that only defines a set of additional methods that a class should provide. Mix-in classes don’t define their own instance attributes nor require their `__init__` constructor to be called.

Writing mix-ins is easy because Python makes it trivial to inspect the current state of any object regardless of its type. Dynamic inspection lets you write generic functionality a single time, in a mix-in, that can be applied to many other classes. Mix-ins can be composed and layered to minimize repetitive code and maximize reuse.

For example, say you want the ability to convert a Python object from its in-memory representation to a dictionary that’s ready for serialization. Why not write this functionality generically so you can use it with all of your classes?

Here, I define an example mix-in that accomplishes this with a new public method that’s added to any class that inherits from it:

```
class ToDictMixin(object):
    def to_dict(self):
        return self._traverse_dict(self.__dict__)
```

The implementation details are straightforward and rely on dynamic attribute access using `hasattr`, dynamic type inspection with `isinstance`, and accessing the instance dictionary `__dict__`.

```
def _traverse_dict(self, instance_dict): output = {}
for key, value in instance_dict.items(): output[key] = self._traverse(key, value)
return output

def _traverse(self, key, value):

if isinstance(value, ToDictMixin): return value.to_dict()
elif isinstance(value, dict):

return self._traverse_dict(value) elif isinstance(value, list):
return [self._traverse(key, i) for i in value] elif hasattr(value, '__dict__'):
return self._traverse_dict(value.__dict__) else:
return value
```

Here, I define an example class that uses the mix-in to make a dictionary representation of a binary tree:

```
class BinaryTree(ToDictMixin):

def __init__(self, value, left=None, right=None): self.value = value
self.left = left self.right = right
```

Translating a large number of related Python objects into a dictionary becomes easy.

```
tree = BinaryTree(10,

left=BinaryTree(7, right=BinaryTree(9)), right=BinaryTree(13, left=BinaryTree(11)))
print(tree.to_dict())

>>>

{'left': {'left': None,

'right': {'left': None, 'right': None, 'value': 9}, 'value': 7},
'right': {'left': {'left': None, 'right': None, 'value': 11}, 'right': None,
'value': 13}, 'value': 10}
```

The best part about mix-ins is that you can make their generic functionality pluggable so behaviors can be overridden when required. For example, here I define a subclass of `BinaryTree` that holds a reference to its parent. This circular reference would cause the default implementation of `ToDictMixin.to_dict` to loop forever.

```
class BinaryTreeWithParent(BinaryTree): def __init__(self, value, left=None,
right=None, parent=None): super().__init__(value, left=left, right=right) self.parent = p
```

The solution is to override the `ToDictMixin._traverse` method in the `BinaryTreeWithParent` class to only process values that matter, preventing cycles encountered by the mix-in. Here, I override the `_traverse` method to not traverse the parent and just insert its numerical value:

```
def _traverse(self, key, value):

if (isinstance(value, BinaryTreeWithParent) and key == 'parent'):
return value.value # Prevent cycles else:
return super()._traverse(key, value)
```

Calling `BinaryTreeWithParent.to_dict` will work without issue because the circular referencing properties aren't followed.

```
root = BinaryTreeWithParent(10)

root.left = BinaryTreeWithParent(7, parent=root) root.left.right = BinaryTreeWithParent(9
>>>

{'left': {'left': None, 'parent': 10,
'right': {'left': None, 'parent': 7, 'right': None, 'value': 9},
'value': 7}, 'parent': None, 'right': None, 'value': 10}
```

By defining `BinaryTreeWithParent._traverse`, I've also enabled any class that has an attribute of type `BinaryTreeWithParent` to automatically work with `ToDictMixin`.

```

class NamedSubTree(ToDictMixin):

    def __init__(self, name, tree_with_parent): self.name = name
    self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right) print(my_tree.to_dict())    # No infinity

>>>

{'name': 'foobar', 'tree_with_parent': {'left': None,
    'parent': 7, 'right': None, 'value': 9}}

```

Mix-ins can also be composed together. For example, say you want a mix-in that provides generic JSON serialization for any class. You can do this by assuming that a class provides a `to_dict` method (which may or may not be provided by the `ToDictMixin` class).

```

class JsonMixin(object): @classmethod
    def from_json(cls, data): kwargs = json.loads(data) return cls(**kwargs)

    def to_json(self):

        return json.dumps(self.to_dict())

```

Note how the `JsonMixin` class defines both instance methods and class methods. Mix-ins let you add either kind of behavior. In this example, the only requirements of the `JsonMixin` are that the class has a *to_dict* method and its *__init__* method takes keyword arguments (see Item 19: “Provide Optional Behavior with Keyword Arguments”).

This mix-in makes it simple to create hierarchies of utility classes that can be serialized to and from JSON with little boilerplate. For example, here I have a hierarchy of data classes representing parts of a datacenter topology:

```

class DatacenterRack(ToDictMixin, JsonMixin):

    def __init__(self, switch=None, machines=None): self.switch = Switch(**switch) self.machines = [Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin): # ...

class Machine(ToDictMixin, JsonMixin): # ...

```

Serializing these classes to and from JSON is simple. Here, I verify that the data is able to be sent round-trip through serializing and deserializing:

```
serialized = """{  
  
  "switch": {"ports": 5, "speed": 1e9}, "machines": [  
    {"cores": 8, "ram": 32e9, "disk": 5e12}, {"cores": 4, "ram": 16e9, "disk": 1e12}, {"cores  
  ] }"""  
  
deserialized = DatacenterRack.from_json(serialized) roundtrip = deserialized.to_json()  
assert json.loads(serialized) == json.loads(roundtrip)
```

When you use mix-ins like this, it's also fine if the class already inherits from `JsonMixin` higher up in the object hierarchy. The resulting class will behave the same way.

Things to Remember

Avoid using multiple inheritance if mix-in classes can achieve the same outcome.

Use pluggable behaviors at the instance level to provide per-class customization when mix-in classes may require it.

Compose mix-ins to create complex functionality from simple behaviors.

Item 27: Prefer Public Attributes Over Private Ones

In Python, there are only two types of attribute visibility for a class's attributes: public and private.

```
class MyObject(object):
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

Public attributes can be accessed by anyone using the dot operator on the object.

```
foo = MyObject()

assert foo.public_field == 5
```

Private fields are specified by prefixing an attribute's name with a double underscore. They can be accessed directly by methods of the containing class.

```
assert foo.get_private_field() == 10
```

Directly accessing private fields from outside the class

```
foo.__private_field

>>>

AttributeError: 'MyObject' object has no attribute '__private_field'
```

Class methods also have access to private attributes because they are declared within the surrounding class block.


```
class MyOtherObject(object):
    def __init__(self):
        self.__private_field = 71

    @classmethod
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field

bar = MyOtherObject()

assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

As you'd expect with private fields, a subclass can't access its parent class's private fields.

```
class MyParentObject(object):
    def __init__(self):
        self.__private_field = 71

class MyChildObject(MyParentObject):
    def get_private_field(self):
        return self.__private_field

baz = MyChildObject()
baz.get_private_field()

>>>

AttributeError: 'MyChildObject' object has no attribute '_MyChildObject__private_field'
```

The private attribute behavior is implemented with a simple transformation of the attribute name. When the Python compiler sees private attribute access in methods like `MyChildObject.getprivatefield`, it translates ***private_field*** to access ***__MyChildObjectprivate_field*** instead. In this example, ***private_field*** was only defined in ***MyParentObject.__init__***, meaning the private attribute's real name is `__MyParentObject__private_field`. Accessing the parent's private attribute from the child class fails simply because the transformed attribute name doesn't match.

Knowing this scheme, you can easily access the private attributes of any class, from a subclass or externally, without asking for permission.

```
assert baz.__MyParentObject__private_field == 71
```

If you look in the object's attribute dictionary, you'll see that private attributes are actually stored with the names as they appear after the transformation.

```
print(baz.__dict__)

>>>

{'_MyParentObject__private_field': 71}
```

Why doesn't the syntax for private attributes actually enforce strict visibility? The simplest answer is one often-quoted motto of Python: "We are all consenting adults here." Python programmers believe that the benefits of being open outweigh the downsides of being closed.

Beyond that, having the ability to hook language features like attribute access (see Item 32: "Use **getattr**, **getattr**, and **setattr** for Lazy Attributes") enables you to mess around with the internals of objects whenever you wish. If you can do that, what is the value of Python trying to prevent private attribute access otherwise?

To minimize the damage of accessing internals unknowingly, Python programmers follow a naming convention defined in the style guide (see Item 2: "Follow the PEP8 Style Guide"). Fields prefixed by a single underscore (like `_protected_field`) are protected, meaning external users of the class should proceed with caution.

However, many programmers who are new to Python use private fields to indicate an internal API that shouldn't be accessed by subclasses or externally.

```
class MyClass(object):

    def __init__(self, value): self.__value = value

    def get_value(self):

        return str(self.__value)

foo = MyClass(5)

assert foo.get_value() == '5'
```

This is the wrong approach. Inevitably someone, including you, will want to subclass your class to add new behavior or to work around deficiencies in existing methods (like above, how `MyClass.get_value` always returns a string). By choosing private attributes, you're only making subclass overrides and extensions cumbersome and brittle. Your potential subclassers will still access the private fields when they absolutely need to do so.

```
class MyIntegerSubclass(MyClass): def get_value(self):
    return int(self._MyClass__value)

foo = MyIntegerSubclass(5) assert foo.get_value() == 5
```

But if the class hierarchy changes beneath you, these classes will break because the private references are no longer valid. Here, the `MyIntegerSubclass` class's immediate parent, `MyClass`, has had another parent class added called `MyBaseClass`:

```
class MyBaseClass(object):

    def __init__(self, value): self.__value = value
    # ...

class MyClass(MyBaseClass): # ...

class MyIntegerSubclass(MyClass): def get_value(self):
    return int(self._MyClass__value)
```

The **value attribute is now assigned in the `MyBaseClass` parent class, not the `MyClass` parent. That causes the private variable reference `self._MyClass__value` to break in `MyIntegerSubclass`.**

```
foo = MyIntegerSubclass(5) foo.get_value()

>>>

AttributeError: 'MyIntegerSubclass' object has no attribute '_MyClass__value'
```

In general, it's better to err on the side of allowing subclasses to do more by using protected attributes. Document each protected field and explain which are internal APIs available to subclasses and which should be left alone entirely. This is as much advice to other programmers as it is guidance for your future self on how to extend your own code safely.

```
class MyClass(object):  
  
    def __init__(self, value):  
  
        # This stores the user-supplied value for the object.  
  
        # It should be coercible to a string. Once assigned for # the object it should be treated
```

The only time to seriously consider using private attributes is when you're worried about naming conflicts with subclasses. This problem occurs when a child class unwittingly defines an attribute that was already defined by its parent class.

```
class ApiClass(object):  
    def __init__(self): self._value = 5  
  
    def get(self):  
  
        return self._value  
  
class Child(ApiClass):  
    def __init__(self):  
        super().__init__()  
  
        self._value = 'hello'      # Conflicts  
  
a = Child()  
  
print(a.get(), 'and', a._value, 'should be different')  
  
>>>  
  
hello and hello should be different
```

This is primarily a concern with classes that are part of a public API; the subclasses are out of your control, so you can't refactor to fix the problem. Such a conflict is especially possible with attribute names that are very common (like `value`). To reduce the risk of this happening, you can use a private attribute in the parent class to ensure that there are no attribute names that overlap with child classes.

```
class ApiClass(object): def __init__(self):
    self.__value = 5

    def get(self):

        return self.__value

class Child(ApiClass): def __init__(self):
    super().__init__() self.__value = 'hello'    # OK!

a = Child()

print(a.get(), 'and', a.__value, 'are different')

>>>

5 and hello are different
```

Things to Remember

Private attributes aren't rigorously enforced by the Python compiler.

Plan from the beginning to allow subclasses to do more with your internal APIs and attributes instead of locking them out by default.

Use documentation of protected fields to guide subclasses instead of trying to force access control with private attributes.

Only consider using private attributes to avoid naming conflicts with subclasses that are out of your control.

Item 28: Inherit from `collections.abc` for Custom Container Types

Much of programming in Python is defining classes that contain data and describing how such objects relate to each other. Every Python class is a container of some kind, encapsulating attributes and functionality together. Python also provides built-in container types for managing data: lists, tuples, sets, and dictionaries.

When you're designing classes for simple use cases like sequences, it's natural that you'd want to subclass Python's built-in list type directly. For example, say you want to create your own custom list type that has additional methods for counting the frequency of its members.

```
class FrequencyList(list):  
  
    def __init__(self, members): super().__init__(members)  
  
    def frequency(self): counts = {}  
    for item in self: counts.setdefault(item, 0) counts[item] += 1  
    return counts
```

By subclassing list, you get all of list's standard functionality and preserve the semantics familiar to all Python programmers. Your additional methods can add any custom behaviors you need.

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd']) print('Length is', len(foo))  
foo.pop()  
  
print('After pop:', repr(foo)) print('Frequency:', foo.frequency())  
  
>>>  
  
Length is 7  
  
After pop: ['a', 'b', 'a', 'c', 'b', 'a'] Frequency: {'a': 3, 'c': 1, 'b': 2}
```

Now, imagine you want to provide an object that feels like a list, allowing indexing, but isn't a list subclass. For example, say you want to provide sequence semantics (like list or tuple) for a binary tree class.

```
class BinaryNode(object):

    def __init__(self, value, left=None, right=None): self.value = value
    self.left = left self.right = right
```

How do you make this act like a sequence type? Python implements its container behaviors with instance methods that have special names. When you access a sequence item by index: `bar = [1, 2, 3]` `bar[0]`

it will be interpreted as: `bar.getitem(0)`

To make the `BinaryNode` class act like a sequence, you can provide a custom implementation of **`getitem`** that traverses the object tree depth first.

```
class IndexableNode(BinaryNode):

    def _search(self, count, index): # ...
    # Returns (found, count)

    def __getitem__(self, index):

        found, _ = self._search(0, index) if not found:
            raise IndexError('Index out of range') return found.value
```

You can construct your binary tree as usual.

```
tree = IndexableNode( 10, left=IndexableNode(
5, left=IndexableNode(2), right=IndexableNode(
6, right=IndexableNode(7))), right=IndexableNode(
15, left=IndexableNode(11)))
```

But you can also access it like a list in addition to tree traversal.

```

print('LRR =', tree.left.right.right.value) print('Index 0 =', tree[0])
print('Index 1 =', tree[1]) print('11 in the tree?', 11 in tree) print('17 in the tree?',

>>>

LRR = 7 Index 0 = 2 Index 1 = 5
11 in the tree? True 17 in the tree? False
Tree is [2, 5, 6, 7, 10, 11, 15]

```

The problem is that implementing **getitem** isn't enough to provide all of the sequence semantics you'd expect.

```

len(tree)

>>>
TypeError: object of type 'IndexableNode' has no len()

```

The `len` built-in function requires another special method named **len** that must have an implementation for your custom sequence type.

```

class SequenceNode(IndexableNode):
    def __len__(self):
        _, count = self._search(0, None)
        return count

tree = SequenceNode( # ...
)

print('Tree has %d nodes' % len(tree))

>>>

Tree has 7 nodes

```

Unfortunately, this still isn't enough. Also missing are the count and index methods that a Python programmer would expect to see on a sequence like list or tuple. Defining your own container types is much harder than it looks.

To avoid this difficulty throughout the Python universe, the built-in `collections.abc` module defines a set of abstract base classes that provide all of the typical methods for each container type. When you subclass from these abstract base classes and forget to implement required methods, the module will tell you something is wrong.



```
from collections.abc import Sequence
```

```
class BadType(Sequence): pass
```

```
foo = BadType()
```

```
>>>
```

```
TypeError: Can't instantiate abstract class BadType with abstract methods __getitem__, __
```

A screenshot of a terminal window with a light gray background. It shows a Python error message: "TypeError: Can't instantiate abstract class BadType with abstract methods __getitem__, __". The text is in a monospaced font. Below the text is a horizontal scrollbar with a small square handle on the left and a small square handle on the right.

When you do implement all of the methods required by an abstract base class, as I did above with `SequenceNode`, it will provide all of the additional methods like `index` and `count` for free.

```
class BetterNode(SequenceNode, Sequence): pass
```

```
tree = BetterNode( # ...  
)
```

```
print('Index of 7 is', tree.index(7)) print('Count of 10 is', tree.count(10))
```

```
>>>
```

```
Index of 7 is 3 Count of 10 is 1
```

The benefit of using these abstract base classes is even greater for more complex types like `Set` and `MutableMapping`, which have a large number of special methods that need to be implemented to match Python conventions.

Things to Remember

Inherit directly from Python's container types (like `list` or `dict`) for simple use cases.

Beware of the large number of methods required to implement custom container types correctly.

Have your custom container types inherit from the interfaces defined in `collections.abc` to ensure that your classes match required interfaces and behaviors.

Concurrency and Parallelism

Concurrency is when a computer does many different things seemingly at the same time. For example, on a computer with one CPU core, the operating system will rapidly change which program is running on the single processor. This interleaves execution of the programs, providing the illusion that the programs are running simultaneously.

Parallelism is actually doing many different things at the same time. Computers with multiple CPU cores can execute multiple programs simultaneously. Each CPU core runs the instructions of a separate program, allowing each program to make forward progress during the same instant.

Within a single program, concurrency is a tool that makes it easier for programmers to solve certain types of problems. Concurrent programs enable many distinct paths of execution to make forward progress in a way that seems to be both simultaneous and independent.

The key difference between parallelism and concurrency is speedup. When two distinct paths of execution in a program make forward progress in parallel, the time it takes to do the total work is cut in half; the speed of execution is faster by a factor of two. In contrast, concurrent programs may run thousands of separate paths of execution seemingly in parallel but provide no speedup for the total work.

Python makes it easy to write concurrent programs. Python can also be used to do parallel work through system calls, subprocesses, and C-extensions. But it can be very difficult to make concurrent Python code truly run in parallel. It's important to understand how to best utilize Python in these subtly different situations.

Built-in Modules

Python takes a “batteries included” approach to the standard library. Many other languages ship with a small number of common packages and require you to look elsewhere for important functionality. Although Python also has an impressive repository of communitybuilt modules, it strives to provide, in its default installation, the most important modules for common uses of the language.

The full set of standard modules is too large to cover in this book. But some of these builtin packages are so closely intertwined with idiomatic Python that they may as well be part of the language specification. These essential built-in modules are especially important when writing the intricate, error-prone parts of programs.

Item 42: Define Function Decorators with `functools.wraps`

Python has special syntax for decorators that can be applied to functions. Decorators have the ability to run additional code before and after any calls to the functions they wrap. This allows them to access and modify input arguments and return values. This functionality can be useful for enforcing semantics, debugging, registering functions, and more.

For example, say you want to print the arguments and return value of a function call. This is especially helpful when debugging a stack of function calls from a recursive function. Here, I define such a decorator:

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print('%s(%r, %r) -> %r' %
              (func.__name__, args, kwargs, result))
        return result
    return wrapper
```

I can apply this to a function using the `@` symbol.

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

The `@` symbol is equivalent to calling the decorator on the function it wraps and assigning the return value to the original name in the same scope.

```
fibonacci = trace(fibonacci)
```

Calling this decorated function will run the wrapper code before and after `fibonacci` runs, printing the arguments and return value at each level in the recursive stack.

```

fibonacci(3)
>>>
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((3,), {}) -> 2

```

This works well, but it has an unintended side effect. The value returned by the decorator — the function that’s called above—doesn’t think it’s named `fibonacci`.

```

print(fibonacci)
>>>
<function trace.<locals>.wrapper at 0x107f7ed08>

```

The cause of this isn’t hard to see. The `trace` function returns the wrapper it defines. The wrapper function is what’s assigned to the `fibonacci` name in the containing module because of the decorator. This behavior is problematic because it undermines tools that do introspection, such as debuggers (see Item 57: “Consider Interactive Debugging with `pdb`”) and object serializers (see Item 44: “Make pickle Reliable with `copyreg`”).

For example, the `help` built-in function is useless on the decorated `fibonacci` function.

```

help(fibonacci)
>>>
Help on function wrapper in module __main__:
wrapper(*args, **kwargs)

```

The solution is to use the `wraps` helper function from the `functools` built-in module. This is a decorator that helps you write decorators. Applying it to the wrapper function will copy all of the important metadata about the inner function to the outer function.

```

def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # ...
    return wrapper
@trace
def fibonacci(n):
    # ...

```

Now, running the `help` function produces the expected result, even though the function is decorated.

```
help(fibonacci)
>>>
Help on function fibonacci in module __main__:
fibonacci(n)
    Return the n-th Fibonacci number
```

Calling help is just one example of how decorators can subtly cause problems. Python functions have many other standard attributes (e.g., `__name__`, `__module__`) that must be preserved to maintain the interface of functions in the language. Using wraps ensures that you'll always get the correct behavior.

Things to Remember

- Decorators are Python syntax for allowing one function to modify another function at runtime.
- Using decorators can cause strange behaviors in tools that do introspection, such as debuggers.
- Use the wraps decorator from the functools built-in module when you define your own decorators to avoid any issues.

Item 43: Consider contextlib and with Statements for Reusable try/finally Behavior

The `with` statement in Python is used to indicate when code is running in a special context. For example, mutual exclusion locks (see Item 38: “Use Lock to Prevent Data Races in Threads”) can be used in `with` statements to indicate that the indented code only runs while the lock is held.

```
lock = Lock()
with lock:
    print('Lock is held')
```

The example above is equivalent to this **try/finally** construction because the **Lock** class properly enables the **with** statement.

```
lock.acquire()
try:
    print('Lock is held')
finally:
    lock.release()
```

The **with** statement version of this is better because it eliminates the need to write the repetitive code of the **try/finally** construction. It’s easy to make your objects and functions capable of use in **with** statements by using the **contextlib** built-in module. This module contains the **contextmanager** decorator, which lets a simple function be used in **with** statements. This is much easier than defining a new class with the special methods `__enter__` and `__exit__` (the standard way).

For example, say you want a region of your code to have more debug logging sometimes. Here, I define a function that does logging at two severity levels:

```
def my_function():
    logging.debug('Some debug data')
    logging.error('Error log here')
    logging.debug('More debug data')
```

The default log level for my program is **WARNING**, so only the error message will print to screen when I run the function.

```
my_function()  
>>>  
Error log here
```

I can elevate the log level of this function temporarily by defining a context manager. This helper function boosts the logging severity level before running the code in the **with** block and reduces the logging severity level afterward.

```
@contextmanager  
def debug_logging(level):  
    logger = logging.getLogger()  
    old_level = logger.getEffectiveLevel()  
    logger.setLevel(level)  
    try:  
        yield  
    finally:  
        logger.setLevel(old_level)
```

The **yield** expression is the point at which the **with** block's contents will execute. Any exceptions that happen in the **with** block will be re-raised by the **yield** expression for you to catch in the helper function (see Item 40: "Consider Coroutines to Run Many Functions Concurrently" for an explanation of how that works).

Now, I can call the same logging function again, but in the **debug_logging** context. This time, all of the debug messages are printed to the screen during the **with** block. The same function running outside the **with** block won't print debug messages.

```
with debug_logging(logging.DEBUG):  
    print('Inside:')  
    my_function()  
print('After:')  
my_function()  
  
>>>  
Inside:  
Some debug data  
Error log here  
More debug data  
After:  
Error log here
```

Using with Targets

The context manager passed to a **with** statement may also return an object. This object is assigned to a local variable in the **as** part of the compound statement. This gives the code running in the **with** block the ability to directly interact with its context.

For example, say you want to write a file and ensure that it's always closed correctly. You can do this by passing **open** to the **with** statement. **open** returns a file handle for the **as** target of **with** and will close the handle when the **with** block exits.

```
with open('/tmp/my_output.txt', 'w') as handle:
    handle.write('This is some data!')
```

This approach is preferable to manually opening and closing the file handle every time. It gives you confidence that the file is eventually closed when execution leaves the **with** statement. It also encourages you to reduce the amount of code that executes while the file handle is open, which is good practice in general.

To enable your own functions to supply values for **as** targets, all you need to do is **yield** a value from your context manager. For example, here I define a context manager to fetch a **Logger** instance, set its level, and then **yield** it for the **as** target.

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield logger
    finally:
        logger.setLevel(old_level)
```

Calling logging methods like **debug** on the **as** target will produce output because the logging severity level is set low enough in the **with** block. Using the **logging** module directly won't print anything because the default logging severity level for the default program logger is **WARNING**.

```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug('This is my message!')
    logging.debug('This will not print')

>>>
This is my message!
```

After the **with** statement exits, calling debug logging methods on the **Logger** named 'my-log' will not print anything because the default logging severity level has been restored. Error log messages will always print.

```
logger = logging.getLogger('my-log')
logger.debug('Debug will not print')
logger.error('Error will print')

>>>
Error will print
```

Things to Remember

- The **with** statement allows you to reuse logic from try/finally blocks and reduce visual noise.
- The **contextlib** built-in module provides a **contextmanager** decorator that makes it easy to use your own functions in **with** statements.
- The value yielded by context managers is supplied to the **as** part of the **with** statement. It's useful for letting your code directly access the cause of the special context.