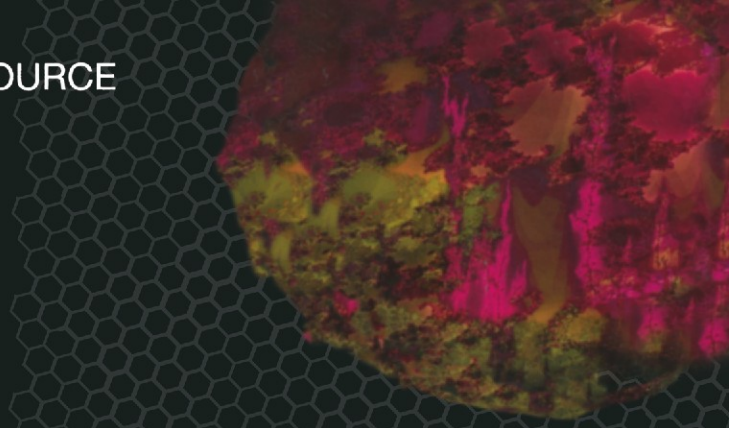
THE EXPERT'S VOICE® IN OPEN SOURCE



Pro Python

Advanced coding techniques and tools

Marty Alchin

Apress®

Pro Python

Marty Alchin

Apress®

Pro Python

ISBN-13 (pbk): 978-1-4302-2757-1

ISBN-13 (electronic): 978-1-4302-2758-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

President and Publisher: Paul Manning
Lead Editors: Duncan Parkes, Tom Welsh
Technical Reviewer: George Vilches
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Mary Tobin
Copy Editors: Nancy Sixsmith, Angel Alchin
Compositor: Bytheway Publishing Services
Indexer: John Collin
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/info/bulksales.

The source code for this book is available to readers at www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

Contents at a Glance

Contents

About the Author

■ **Marty Alchin** is a professional programmer with a passion for the Web. His work with Django, the popular Web framework, led him to write several articles about Python, to speak at PyCon and even to write his first book, *Pro Django*, which was published by Apress in December of 2008.

In addition to writing for print, Marty keeps a blog at `http://martyalchin.com/`, where he writes about Python, Django and anything else that strikes his fancy.

About the Technical Reviewer

George Vilches is a software engineer and systems administrator with an unabashed fondness for Python and the Web in both disciplines. In the last three years, he has made several contributions to Django, with a focus on the ORM and administrative side of things. He is a principal engineer with AOL and builds Django applications with Fortune Cookie Studios (`http://fcstudios.com`).

George's personal time is split between tinkering with open source projects and enjoying the company of his wife Kate, their corgi and their two cats (all of whom would prefer he stop tinkering and attend to them more).

Acknowledgments

I wouldn't have even started this project if not for the endless encouragement from my lovely wife, Angel. She's been my sounding board, my task manager, my copy editor and my own personal cheerleader. There's no way I could do anything like this without her help and support.

I'd also like to thank my technical reviewer, George, for everything he's done to help me out. He's gone above and beyond the limits of his role, helping with everything from code to grammar and even a good bit of style. After enjoying his help on *Pro Django*, I wouldn't have even signed on for another book without him by my side.

Lastly, I never would've considered a book like this if not for the wonderful community around Python. The willingness of Python programmers to open their minds and their code is, I believe, unrivaled among our peers. It's this spirit of openness that encourages me every day, leading me to discover new things and push myself beyond the limits of what I knew yesterday.

We learn by doing and by seeing what others have done. I hope that you'll take the contents of this book and do more with it than what I've done. There's no better reward for all this hard work than to see better programmers writing better code.

Introduction

When I wrote my first book, *Pro Django*, I didn't have much of an idea what my readers would find interesting. I had gained a lot of information I thought would be useful for others to learn, but I didn't really know what would be the most valuable thing they'd take away. As it turned out, in nearly 300 pages, the most popular chapter in the book barely mentioned Django at all. It was about Python.

The response was overwhelming. There was clearly a desire to learn more about how to go from a simple Python application to a detailed framework like Django. It's all Python code, but it can be hard to understand based on even a reasonably thorough understanding of the language. The tools and techniques involved require some extra knowledge that you might not run into in general use.

This gave me a new goal with *Pro Python*: to take you from proficient to professional. Being a true professional requires more experience than you can get from a book, but I want to at least give you the tools you'll need. Combined with the rich philosophy of the Python community, you'll find plenty of information to take your code to the next level.

Who This Book Is For

Because my goal is to bring intermediate programmers to a more advanced level, I wrote this book with the expectation that you'll already be familiar with Python. You should be comfortable using the interactive interpreter, writing control structures and a basic object-oriented approach.

That's not a very difficult prerequisite. If you've tried your hand at writing a Python application—even if you haven't released it into the wild, or even finished it—you likely have all the necessary knowledge to get started. The rest of the information you'll need is contained in these pages.

What You'll Need

This book is written with the latest versions of Python in mind, so most of the examples assume that you're already using Python 3.1, which is the latest official release as of the date of publishing. I don't take the jump to Python 3 lightly, though, so there are plenty of compatibility notes along the way, going all the way back to Python 2.5. As long as your copy of Python was released in the last few years, you'll be all set.

Nearly all the packages used in this book come from the Python Standard Library, which ships with every Python installation. Some sections will reference third-party libraries that aren't included in that bundle, but those are strictly informative; you won't lose out if you don't have them installed.

Source Code

The code for all the examples in this book is available at `http://propython.com/`.

CHAPTER 1

Principles and Philosophy

If it seems strange to begin a programming book with a chapter about philosophy, that's actually evidence of why this chapter is so important. Python was created to embody and encourage a certain set of ideals that have helped guide the decisions of its maintainers and its community for nearly 20 years. Understanding these concepts will help you make the most out of what the language and its community have to offer.

Of course, we're not talking about Plato or Nietzsche here. Python deals with programming problems, and its philosophies are designed to help build reliable, maintainable solutions. Some of these philosophies are officially branded into the Python landscape, while others are guidelines commonly accepted by Python programmers, but all of them will help you write code that is powerful, easy to maintain and understandable to other programmers.

The philosophies laid out in this chapter can be read from start to finish here, but don't expect to commit them all to memory in one pass. The rest of this book will refer back here often, by illustrating which concepts come into play in various situations. After all, the real value of philosophy is understanding how to apply it when it matters most.

The Zen of Python

Perhaps the best known collection of Python philosophy was written by Tim Peters, long-time contributor to the language and its newsgroup, `comp.lang.python`.[1] This Zen of Python condenses some of the most common philosophical concerns into a brief list that's been recorded as both its own Python Enhancement Proposal (PEP)[2] and within Python itself. Something of an easter egg, Python includes a module called `this`.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

[1] http://propython.com/comp-lang-python/
[2] http://propython.coms/pep-20/

```
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

This list was primarily intended as a humorous accounting of Python philosophy, but over the years, numerous Python applications have used these guidelines to greatly improve the quality, readability and maintainability of their code. Just listing the Zen of Python is of little value, though, so the following sections will explain each idiom in more detail.

Beautiful Is Better Than Ugly

Perhaps it's fitting that this first notion is arguably the most subjective of the whole bunch. After all, beauty is in the eye of the beholder, a fact which has been discussed for centuries. It serves as a blatant reminder that philosophy is far from absolute. Still, having something like this in writing provides a goal to strive for, which is the ultimate purpose of all these ideals.

One obvious application of this philosophy is in Python's own language structure, which minimizes the use of punctuation, instead preferring English words where appropriate. Another advantage is Python's focus on keyword arguments, which help clarify function calls that would otherwise be difficult to understand. Consider the following two possible ways of writing the same code, and consider which one looks more beautiful.

```
is_valid = form != null && form.is_valid(true)
is_valid = form is not None and form.is_valid(include_hidden_fields=True)
```

The second example reads a bit more like natural English, and explicitly including the name of the argument gives greater insight into its purpose. In addition to language concerns, coding style can be influenced by similar notions of beauty. The name `is_valid`, for example, asks a simple question, which the method can then be expected to answer with its return value. A name like `validate` would've been ambiguous because it would be an accurate name even if no value were returned at all.

Being so subjective, however, it's dangerous to rely too heavily on beauty as a criterion for a design decision. If other ideals have been considered and you're still left with two workable options, certainly consider factoring beauty into the equation, but do make sure that other facets are taken into account first. You'll likely find a good choice using some of the other criteria long before reaching this point.

Explicit Is Better Than Implicit

Though this notion may seem easier to interpret than beauty, it's actually one of the trickier guidelines to follow. On the surface, it seems simple enough: don't do anything the programmer didn't explicitly command. Beyond just Python itself, frameworks and libraries have a similar responsibility because their code will be accessed by other programmers whose goals will not always be known in advance.

Unfortunately, truly explicit code must account for every nuance of a program's execution, from memory management to display routines. Some programming languages do expect that level of detail from their programmers, but Python doesn't. In order to make the programmer's job easier and allow you to focus on the problem at hand, there need to be some trade-offs along the way.

In general, Python asks you to declare your intentions explicitly, rather than issue every command necessary to make that intention a reality. For example, when assigning a value to a variable, you don't need to worry about setting aside the necessary memory, assigning a pointer to the value and cleaning up the memory once it's no longer in use. Memory management is a necessary part of variable assignment, so Python takes care of it behind the scenes. Assigning the value is enough of an explicit declaration of intent to justify the implicit behavior.

On the other hand, regular expressions in the Perl programming language automatically assign values to special variables any time a match is found. Someone unfamiliar with the way Perl handles that situation wouldn't understand a code snippet that relies on it because variables would seem to come from thin air, with no assignments related to them. Python programmers try to avoid this type of implicit behavior in favor of more readable code.

Because different applications will have different ways of declaring intentions, no single generic explanation will apply to all cases. Instead, this guideline will come up quite frequently throughout the book, clarifying how it would be applied to various situations.

Simple Is Better Than Complex

This is a considerably more concrete guideline, with implications primarily in the design of interfaces to frameworks and libraries. The goal here is to keep the interface as straightforward as possible, leveraging a programmer's knowledge of existing interfaces as much as possible. For example, a caching framework could use the same interface as standard dictionaries rather than inventing a whole new set of method calls.

Of course, there are many other applications of this rule, such as taking advantage of the fact that most expressions can evaluate to true or false without explicit tests. For example, the following two lines of code are functionally identical for strings, but notice the difference in complexity between them.

```
if value is not None and value != '':
if value:
```

As you can see, the second option is much simpler to read and understand. All the situations covered in the first example will evaluate to false anyway, so the simpler test is just as effective. It also has two other benefits: it runs faster, having fewer tests to perform, and it also works in more cases, because individual objects can define their own method of determining whether they should evaluate to true or false.

It may seem like this is something of a convoluted example, but it's just the type of thing that comes up quite frequently. By relying on simpler interfaces, you can often take advantage of optimizations and increased flexibility while producing more readable code.

Complex Is Better Than Complicated

Sometimes, however, a certain level of complexity is required in order to get the job done. Database adapters, for example, don't have the luxury of using a simple dictionary-style interface, but instead require an extensive set of objects and methods to cover all of their features. The important thing to remember in those situations is that complexity doesn't necessarily require it to be complicated.

The tricky bit with this one, obviously, is distinguishing between the two. Dictionary definitions of each term often reference the other, considerably blurring the line between the two. For the sake of this guideline, most situations tend to take the following view of the two terms.

- Complex—made up of many interconnected parts.
- Complicated—so complex as to be difficult to understand.

So in the face of an interface that requires a large number of things to keep track of, it's even more important to retain as much simplicity as possible. This can take the form of consolidating methods onto a smaller number of objects, perhaps grouping objects into more logical arrangements or even simply making sure to use names that make sense without having to dig into the code to understand them.

Flat Is Better Than Nested

This guideline might not seem to make sense at first, but it's about how structures are laid out. The structures in question could be objects and their attributes, packages and their included modules or even code blocks within a function. The goal is to keep things as relationships of peers as much possible, rather than parents and children. For example, take the following code snippet, which illustrates the problem.

```
if x > 0:
    if y > 100:
        raise ValueError("Value for y is too large.")
    else:
        return y
else:
    if x == 0:
        return False
    else:
        raise ValueError("Value for x cannot be negative.")
```

In this example, it's fairly difficult to follow what's really going on because the nested nature of the code blocks requires you to keep track of multiple levels of conditions. Consider the following alternative approach to writing the same code, flattening it out.

```
if x > 0 and y > 100:
    raise ValueError("Value for y is too large.")
elif x > 0:
    return y
elif x == 0:
    return False
else:
    raise ValueError("Value for x cannot be negative.")
```

Notice how much easier it is to follow the logic in the second example because all the conditions are at the same level. It even saves two lines of code by avoiding the extraneous `else` blocks along the way. This is actually the main reason for the existence of the `elif` keyword; Python's use of indentation means that complex `if` blocks can quickly get out of hand otherwise.

■ Caution What might not be as obvious is that the refactoring of this example ends up testing `x > 0` twice, where it was only performed once previously. If that test had been an expensive operation, such as a database query, refactoring it in this way would reduce the performance of the program, so it wouldn't be worth it. This is covered in detail in a later guideline: Practicality Beats Purity.

In the case of package layouts, flat structures can often allow a single import to make the entire package available under a single namespace. Otherwise, the programmer would need to know the full structure in order to find the particular class or function required. Some packages are so complex that a nested structure will help reduce clutter on each individual namespace, but it's best to start flat and nest only when problems arise.

Sparse Is Better Than Dense

This principle largely pertains to the visual appearance of Python source code, favoring the use of whitespace to differentiate among blocks of code. The goal is to keep highly related snippets together, while separating them from subsequent or unrelated code, rather than simply having everything run together in an effort to save a few bytes on disk.

In the real world, there are plenty of specific concerns to address, such as how to separate module-level classes or deal with one-line `if` blocks. Though no single set of rules will be appropriate for all projects, PEP-8[3] does specify many aspects of source code layout that help you adhere to this principle. It provides a number of hints on how to format import statements, classes, functions and even many types of expressions.

It's interesting to note that PEP-8 includes a number of rules about expressions in particular, which specifically encourage avoiding extra spaces. Take the following examples, taken straight from PEP-8.

```
Yes: spam(ham[1], {eggs: 2})
No:  spam( ham[ 1 ], { eggs: 2 } )

Yes: if x == 4: print x, y; x, y = y, x
No:  if x == 4 : print x , y ; x , y = y , x

Yes: spam(1)
No:  spam (1)

Yes: dict['key'] = list[index]
No:  dict ['key'] = list [index]
```

The key to this apparent discrepancy is that whitespace is a valuable resource and should be distributed responsibly. After all, if everything tries to stand out in any one particular way, nothing really does stand out at all. If you use whitespace to separate even highly related bits of code like the above expressions, truly unrelated code isn't any different from the rest.

That's perhaps the most important part of this principle and the key to applying it to other aspects of code design. When writing libraries or frameworks, it's generally better to define a small set of unique types of objects and interfaces that can be reused across the application, maintaining similarity where appropriate and differentiating the rest.

Readability Counts

Finally, we have a principle everybody in the Python world can get behind, but that's mostly because it's one of the most vague in the entire collection. In a way, it sums up the whole of Python philosophy in one deft stroke, but it also leaves so much undefined that it's worth examining a bit further.

Readability covers a wide range of issues, such as the names of modules, classes, functions and variables. It includes the style of individual blocks of code and the whitespace between them. It can even

[3] http://propython.com/pep-8/

pertain to the separation of responsibilities among multiple functions or classes if that separation is done so that it's more readable to the human eye.

That's the real point here: code gets read not only by computers, but also by humans who have to maintain it. Those humans have to read existing code far more often than they have to write new code, and it's often code that was written by someone else. Readability is all about actively promoting human understanding of code.

Development is much easier in the long run when everyone involved can simply open up a file and easily understand what's going on in it. This seems like a given in organizations with high turnover, where new programmers must regularly read the code of their predecessors, but it's true even for those who have to read their own code weeks, months or even years after it was written. Once we lose our original train of thought, all we have to remind us is the code itself, so it's very valuable to take the extra time to make it easy to read.

The best part is how little extra time it often takes. It can be as simple as adding a blank line between two functions or naming variables with nouns and functions with verbs. It's really more of a frame of mind than a set of rules, though. A focus on readability requires you to always look at your code as a human being would, rather than only as a computer would. Remember the Golden Rule: do for others what you'd like them to do for you. Readability is random acts of kindness sprinkled throughout your code.

Special Cases Aren't Special Enough to Break the Rules

Just as "Readability counts" is a banner phrase for how we should approach our code at all times, this principle is about the conviction with which we must pursue it. It's all well and good to get it right most of the time, but all it takes is one ugly chunk of code to undermine all that hard work.

What's perhaps most interesting about this rule, though, is that it doesn't pertain just to readability or any other single aspect of code. It's really just about the conviction to stand behind the decisions you've made, regardless of what those are. If you're committed to backward compatibility, internationalization, readability or anything else, don't break those promises just because a new feature comes along and makes some things a bit easier.

Although Practicality Beats Purity

And here's where things get tricky. The previous principle encourages you to always do the right thing, regardless of how exceptional one situation might be, where this one seems to allow exceptions whenever the right thing gets difficult. The reality is a bit more complicated, though, and merits some discussion.

Up to this point, it seemed simple enough at a glance: the fastest, most efficient code might not always be the most readable, so you may have to accept subpar performance to gain code that's easier to maintain. This is certainly true in many cases, and much of Python's standard library is less than ideal in terms of raw performance, instead opting for pure Python implementations that are more readable and more portable to other environments, like Jython or IronPython. On a larger scale, though, the problem goes deeper than that.

When designing a system at any level, it's easy to get into a heads-down mode, where you focus exclusively on the problem at hand and how best to solve it. This might involve algorithms, optimizations, interface schemes or even refactorings, but it typically boils down to working on one thing so hard that you don't look at the bigger picture for a while. In that mode, programmers commonly do what seems best within the current context, but when backing out a bit for a better look, those decisions don't match up with the rest of the application as a whole.

It's not always easy to know which way to go at this point. Do you try to optimize the rest of the application to match that perfect routine you just wrote? Do you rewrite the otherwise-perfect function in hopes of gaining a more cohesive whole? Or do you just leave the inconsistency alone, hoping it

doesn't trip anybody up? The answer, as usual, depends on the situation, but one of those options will often seem more practical in context than the others.

Typically, it's preferable to maintain greater overall consistency at the expense of a few small areas that may be less than ideal. Again, most of Python's standard library uses this approach, but there are exceptions even there. Packages that require a lot of computational power or get used in applications that need to avoid bottlenecks will often be written in C to improve performance, at the cost of maintainability. These packages then need to be ported over to other environments and tested more rigorously on different systems, but the speed gained serves a more practical purpose than a purer Python implementation would allow.

Errors Should Never Pass Silently

Python supports a robust error-handling system, with dozens of built-in exceptions provided out of the box, but there's often doubt about when those exceptions should be used and when new ones are necessary. The guidance provided by this line of the Zen of Python is quite simple, but as with so many others, there's much more beneath the surface.

The first task is to clarify the definitions of errors and exceptions. Even though these words, like so many others in the world of computing, are often overloaded with additional meaning, there's definite value in looking at them as they're used in general language. Consider the following definitions, as found in the *Merriam-Webster Dictionary*:

- An act or condition of ignorant or imprudent deviation from a code of behavior
- A case to which a rule does not apply

The terms themselves have been left out here to help illustrate just how similar the two definitions can be. In real life, the biggest observed difference between the two terms is the severity of the problems caused by deviations from the norm. Exceptions are typically considered less disruptive and thus more acceptable, but they both amount to the same thing: a violation of some kind of expectation. For the purposes of this discussion, the term exception will be used to refer to any such departure from the norm.

■ **Note** One important thing to realize, though, is that not all exceptions are errors. Some are used to enhance code flow options, such as using `StopIteration`, which is documented in Chapter 5. In code flow usage, exceptions provide a way to indicate what happened inside a function, even though that indication has no relationship to its return value.

This interpretation makes it impossible to describe exceptions on their own; they must be placed in the context of an expectation that can be violated. Every time we write a piece of code, we make a promise that it'll work in a specific way. Exceptions break that promise, so we need to understand what types of promises we make and how they can be broken. Take the following simple Python function and look for any promises that can be broken.

```
def validate(data):
    if data['username'].startswith('_'):
        raise ValueError("Username must not begin with an underscore.")
```

The obvious promise here is that of `validate()` itself: if the incoming data is valid, the function will return silently. Violations of that rule, such as a username beginning with an underscore, are explicitly treated as an exception, neatly illustrating this practice of not allowing errors to pass silently. Raising an exception draws attention to the situation and provides enough information for the code that called this function to understand what happened.

The tricky bit here is to see the other exceptions that may get raised. For example, if the `data` dictionary doesn't contain a `'username'` key, as the function expects, Python will raise a `KeyError`. If that key does exist, but its value isn't a string, Python will raise an `AttributeError` when trying to access the `startswith()` method. If `data` isn't a dictionary at all, Python would raise a `TypeError`.

Most of those assumptions are true requirements for proper operation, but they don't all have to be. Let's assume this validation function could be called from a number of contexts, some of which may not have even asked for a username. In those cases, a missing username isn't actually an exception at all, but just another flow that needs to be accounted for.

With that new requirement in mind, `validate()` can be slightly altered to no longer rely on the presence of a `'username'` key to work properly. All the other assumptions should stay intact, however, and should raise their respective exceptions when violated. Here's how it might look after this change.

```
def validate(data):
    if 'username' in data and data['username'].startswith('_'):
        raise ValueError("Username must not begin with an underscore.")
```

And just like that, one assumption has been removed and the function can now run just fine without a username supplied in the `data` dictionary. Alternately, you could now check for a missing username explicitly and raise a more specific exception if truly required. How the remaining exceptions are handled depends on the needs of the code that calls `validate()`, and there's a complementary principle to deal with that situation.

Unless Explicitly Silenced

Like any other language that supports exceptions, Python allows the code that triggers exceptions to trap them and handle them in different ways. In the preceding validation example, it's likely that the validation errors should be shown to the user in a nicer way than a full traceback. Consider a small command-line program that accepts a username as an argument and validates it against the rules defined previously.

```
import sys

def validate(data):
    if 'username' in data and data['username'].startswith('_'):
        raise ValueError("Username must not begin with an underscore.")

if __name__ == '__main__':
    username = sys.argv[1]
    try:
        validate({'username': username})
    except TypeError, ValueError as e:
        print e
```

Compatibility: Prior to 3.0

The syntax used to catch the exception and store it as the variable `e` in this example was made available in Python 3.0. Previously, the `except` clause used commas to separate exception types from each other and to distinguish the name of the variable to hold the exception, so the example here would read `except (TypeError, ValueError), e`. To resolve this ambiguity, the `as` keyword was added to Python 2.6, which makes blocks like this much more explicit.

The comma syntax will work in all Python versions up to and including 2.7, while Python 2.6 and higher support the as keyword shown here. Python 2.6 and 2.7 support both syntaxes in an effort to ease the transition.

In this example, all those exceptions that might be raised will simply get caught by this code, and the message alone will be displayed to the user, not the full traceback. This form of error handling allows for complex code to use exceptions to indicate violated expectations without taking down the whole program.

Explicit is better than implicit

In a nutshell, this error-handling system is a simple example of the previous rule favoring explicit declarations over implicit behavior. The default behavior is as obvious as possible, given that exceptions always propagate upward to higher levels of code, but can be overridden using an explicit syntax.

In the Face of Ambiguity, Refuse the Temptation to Guess

Sometimes, when using or implementing interfaces between pieces of code written by different people, certain aspects may not always be clear. For example, one common practice is to pass around byte strings without any information about what encoding they rely on. This means that if any code needs to convert those strings to Unicode or ensure that they use a specific encoding, there's not enough information available to do so.

It's tempting to play the odds in this situation, blindly picking what seems to be the most common encoding. Surely it would handle most cases, and that should be enough for any real-world application. Alas, no. Encoding problems raise exceptions in Python, so those could either take down the application or they could be caught and ignored, which could inadvertently cause other parts of the application to think strings were properly converted when they actually weren't.

Worse yet, your application now relies on a guess. It's an educated guess, of course, perhaps with the odds on your side, but real life has a nasty habit of flying in the face of probability. You might well find that what you assumed to be most common is in fact less likely when given real data from real people. Not only could incorrect encodings cause problems with your application, those problems could occur far more frequently than you realize.

A better approach would be to only accept Unicode strings, which can then be written to byte strings using whatever encoding your application chooses. That removes all ambiguity, so your code doesn't have to guess anymore. Of course, if your application doesn't need to deal with Unicode and can simply pass byte strings through unconverted, it should accept byte strings only, rather than you having to guess an encoding to use to produce byte strings.

There Should Be One—and Preferably Only One—Obvious Way to Do It

Though similar to the previous principle, this one is generally applied only to development of libraries and frameworks. When designing a module, class or function, it may be tempting to implement a number of entry points, each accounting for a slightly different scenario. In the byte string example from the previous section, for example, you might consider having one function to handle byte strings and another to handle Unicode strings.

The problem with that approach is that every interface adds a burden on developers who have to use it. Not only are there more things to remember, but it may not always be clear which function to use even when all the options are known. Choosing the right option often comes down to little more than naming, which can sometimes be a guess in and of itself.

In the previous example, the simple solution is to accept only Unicode strings, which neatly avoids other problems, but for this principle, the recommendation is broader. Stick to simpler, more common interfaces like the protocols illustrated in Chapter 5 where you can, adding on only when you have a truly different task to perform.

You might have noticed that Python itself seems to violate this rule sometimes, most notably in its dictionary implementation. The preferred way to access a value is to use the bracket syntax, `my_dict['key']`, but dictionaries also have a `get()` method, which seems to do the exact same thing. Conflicts like this come up fairly frequently when dealing with such an extensive set of principles, but there are often good reasons if you're willing to consider them.

In the dictionary case, it comes back to the notion of raising an exception when a rule is violated. When thinking about violations of a rule, we have to examine the rules implied by these two available access methods. The bracket syntax follows a very basic rule: return the value referenced by the key provided. It's really that simple. Anything that gets in the way of that, such as an invalid key, a missing value or some additional behavior provided by an overridden protocol, results in an exception being raised.

The `get()` method, on the other hand, follows a more complicated set of rules. It checks to see whether the provided key is present in the dictionary; if it is, the associated value is returned. If the key isn't in the dictionary, an alternate value is returned instead. By default, the alternate value is `None`, but that can be overridden by providing a second argument.

By laying out the rules each technique follows, it becomes clearer why there are two different options. Bracket syntax is the common use case, failing loudly in all but the most optimistic situations, while `get()` offers more flexibility for those situations that need it. One refuses to allow errors to pass silently, while the other explicitly silences them. Essentially, providing two options allows dictionaries to satisfy both principles.

More to the point, though, is that the philosophy states there should only be one *obvious* way to do it. Even in the dictionary example, which has two ways to get values, only one—the bracket syntax—is obvious. The `get()` method is available, but it isn't very well known, and it certainly isn't promoted as the primary interface for working with dictionaries. It's okay to provide multiple ways to do something as long as they're for sufficiently different use cases, and the most common use case is presented as the obvious choice.

Although That Way May Not Be Obvious at First Unless You're Dutch

This is a nod to the homeland of Python's creator and Benevolent Dictator for Life, Guido van Rossum. More importantly, though, it's an acknowledgment that not everyone sees things the same way. What seems obvious to one person might seem completely foreign to somebody else, and though there are