O'REILLY®

How to Make Mistakes in Python

Mike Pirnat

How to Make Mistakes in Python

Mike Pirnat

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

How to Make Mistakes in Python
by Mike Pirnat

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editor: Meghan Blanchette
Production Editor: Kristen Brown
Copyeditor: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2015: First Edition

Revision History for the First Edition
2015-09-25: First Release

978-1-491-93447-0

[LSI]

To my daughter, Claire, who enables me to see the world anew, and to my wife, Elizabeth, partner in the adventure of life.

Table of Contents

Introduction

To err is human; to really foul things up requires a computer.
—Bill Vaughan

I started programming with Python in 2000, at the very tail end of The Bubble. In that time, I've…done things. Things I'm not proud of. Some of them simple, some of them profound, all with good intentions. Mistakes, as they say, have been made. Some have been costly, many of them embarrassing. By talking about them, by investigating them, by peeling them back layer by layer, I hope to save you some of the toe-stubbing and face-palming that I've caused myself.

As I've reflected on the kinds of errors I've made as a Python programmer, I've observed that they fall more or less into the categories that are presented here:

Setup
: How an incautiously prepared environment has hampered me.

Silly things
: The trivial mistakes that waste a disproportionate amount of my energy.

Style
: Poor stylistic decisions that impede readability.

Structure
: Assembling code in ways that make change more difficult.

Surprises
: Those sudden shocking mysteries that only time can turn from OMG to LOL.

There are a couple of quick things that should be addressed before we get started.

First, this work does not aim to be an exhaustive reference on potential programming pitfalls—it would have to be much, much longer, and would probably never be complete—but strives instead to be a meaningful tour of the "greatest hits" of my sins.

My experiences are largely based on working with real-world but closed-source code; though authentic examples are used where possible, code samples that appear here may be abstracted and hyperbolized for effect, with variable names changed to protect the innocent. They may also refer to undefined variables or functions. Code samples make liberal use of the ellipsis (…) to gloss over reams of code that would otherwise obscure the point of the discussion. Examples from real-world code may contain more flaws than those under direct examination.

Due to formatting constraints, some sample code that's described as "one line" may appear on more than one line; I humbly ask the use of your imagination in such cases.

Code examples in this book are written for Python 2, though the concepts under consideration are relevant to Python 3 and likely far beyond.

Thanks are due to Heather Scherer, who coordinated this project; to Leonardo Alemeida, Allen Downey, and Stuart Williams, who provided valuable feedback; to Kristen Brown and Sonia Saruba, who helped tidy everything up; and especially to editor Meghan Blanchette, who picked my weird idea over all of the safe ones and encouraged me to run with it.

Finally, though the material discussed here is rooted in my professional life, it should not be construed as representing the current state of the applications I work with. Rather, it's drawn from over 15 years (an eternity on the web!) and much has changed in that time. I'm deeply grateful to my workplace for the opportunity to make mistakes, to grow as a programmer, and to share what I've learned along the way.

With any luck, after reading this you will be in a position to make a more interesting caliber of mistake: with an awareness of what can go wrong, and how to avoid it, you will be freed to make the exciting, messy, significant sorts of mistakes that push the art of programming, or the domain of your work, forward.

I'm eager to see what kind of trouble you'll get up to.

CHAPTER 1

Setup

Mise-en-place is the religion of all good line cooks…
The universe is in order when your station is set up the way you like it:
you know where to find everything with your eyes closed, everything you
need during the course of the shift is at the ready at arm's reach,
your defenses are deployed.
—Anthony Bourdain

There are a couple of ways I've gotten off on the wrong foot by not starting a project with the right tooling, resulting in lost time and plenty of frustration. In particular, I've made a proper hash of several computers by installing packages willy-nilly, rendering my system Python environment a toxic wasteland, and I've continued to use the default Python shell even though better alternatives are available. Modest up-front investments of time and effort to avoid these issues will pay huge dividends over your career as a Pythonista.

Polluting the System Python

One of Python's great strengths is the vibrant community of developers producing useful third-party packages that you can quickly and easily install. But it's not a good idea to just go wild installing everything that looks interesting, because you can quickly end up with a tangled mess where nothing works right.

By default, when you `pip install` (or in days of yore, `easy_install`) a package, it goes into your computer's system-wide `site-packages` directory. Any time you fire up a Python shell or a Python program, you'll be able to import and use that package.

That may feel okay at first, but once you start developing or working with multiple projects on that computer, you're going to eventually have conflicts over package dependencies. Suppose project P1 depends on version 1.0 of library L, and project P2 uses version 4.2 of library L. If both projects have to be developed or deployed on the same machine, you're practically guaranteed to have a bad day due to changes to the library's interface or behavior; if both projects use the same `site-packages`, they cannot coexist! Even worse, on many Linux distributions, important system tooling is written in Python, so getting into this dependency management hell means you can break critical pieces of your OS.

The solution for this is to use so-called virtual environments. When you create a virtual environment (or "virtual env"), you have a separate Python environment outside of the system Python: the virtual environment has its own `site-packages` directory, but shares the standard library and whatever Python binary you pointed it at during creation. (You can even have some virtual environments using Python 2 and others using Python 3, if that's what you need!)

For Python 2, you'll need to install `virtualenv` by running `pip install virtualenv`, while Python 3 now includes the same functionality out-of-the-box.

To create a virtual environment in a new directory, all you need to do is run one command, though it will vary slightly based on your choice of OS (Unix-like versus Windows) and Python version (2 or 3). For Python 2, you'll use:

```
virtualenv <directory_name>
```

while for Python 3, on Unix-like systems it's:

```
pyvenv <directory_name>
```

and for Python 3 on Windows:

```
pyvenv.py <directory_name>
```

NOTE

Windows users will also need to adjust their `PATH` to include the location of their system Python and its scripts; this procedure varies slightly between versions of Windows, and the exact setting depends on the version of Python. For a standard installation of Python 3.4, for example, the `PATH` should include:

```
C:\Python34\;C:\Python34\Scripts\;C:
\Python34\Tools\Scripts
```

This creates a new directory with everything the virtual environment needs: `lib` (`Lib` on Windows) and `include` subdirectories for supporting library files, and a `bin` subdirectory (`Scripts` on Windows) with scripts to manage the virtual environment and a symbolic link to the appropriate Python binary. It also installs the `pip` and `setuptools` modules in the virtual environment so that you can easily install additional packages.

Once the virtual environment has been created, you'll need to navigate into that directory and "activate" the virtual environment by running a small shell script. This script tweaks the environment variables necessary to use the virtual environment's Python and `site-packages`. If you use the Bash shell, you'll run:

```
source bin/activate
```

Windows users will run:

```
Scripts\activate.bat
```

Equivalents are also provided for the Csh and Fish shells on Unix-like systems, as well as PowerShell on Windows. Once activated, the virtual environment is isolated from your system Python—any packages you install are independent from the system Python as well as from other virtual environments.

When you are done working in that virtual environment, the `deactivate` command will revert to using the default Python again.

As you might guess, I used to think that all this virtual environment stuff was too many moving parts, way too complicated, and I would never need to use it. After causing myself significant amounts of pain, I've changed my tune. Installing `virtualenv` for working with Python 2 code is now one of the first things I do on a new computer.

TIP

If you have more advanced needs and find that pip and virtualenv don't quite cut it for you, you may want to consider Conda as an alternative for managing packages and environments. (I haven't needed it; your mileage may vary.)

Using the Default REPL

When I started with Python, one of the first features I fell in love with was the interactive shell, or REPL (short for Read Evaluate Print Loop). By just firing up an interactive shell, I could explore APIs, test ideas, and sketch out solutions, without the overhead of having a larger program in progress. Its immediacy reminded me fondly of my first programming experiences on the Apple II. Nearly 16 years later, I still reach for that same Python shell when I want to try something out…which is a shame, because there are far better alternatives that I should be using instead.

The most notable of these are IPython and the browser-based Jupyter Notebook (formerly known as IPython Notebook), which have spurred a revolution in the scientific computing community. The powerful IPython shell offers features like tab completion, easy and humane ways to explore objects, an integrated debugger, and the ability to easily review and edit the history you've executed. The Notebook takes the shell even further, providing a compelling web browser experience that can easily combine code, prose, and diagrams, and which enables low-friction distribution and sharing of code and data.

The plain old Python shell is an okay starting place, and you can get a lot done with it, as long as you don't make any mistakes. My experiences tend to look something like this:

```
>>> class Foo(object):
...     def __init__(self, x):
...         self.x = x
...     def bar(self):
...         retrun self.x
  File "<stdin>", line 5
    retrun self.x
              ^
SyntaxError: invalid syntax
```

Okay, I can fix that without retyping everything; I just need to go back into history with the up arrow, so that's…

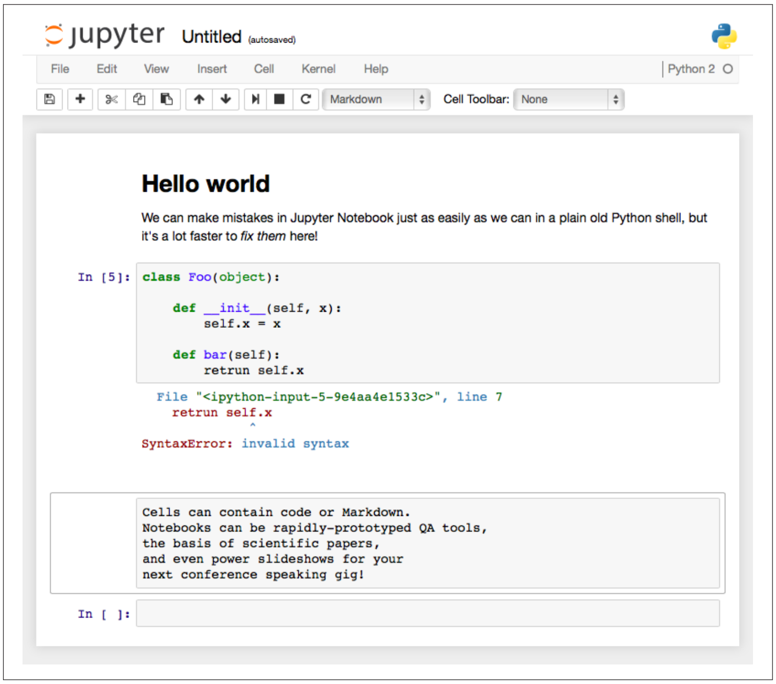
Up arrow. Up. Up. Up. Up. Enter.

Up. Up. Up. Up. Up. Enter. Up. Up. Up. Up. Up. Enter. Up. Up. Up. Up. Up. Enter.

Up. Up. Up. Up. Up. Enter. Then I get the same `SyntaxError` because I got into a rhythm and pressed Enter without fixing the error first. Whoops!

Then I repeat this cycle several times, each iteration punctuated with increasingly sour cursing.

Eventually I'll get it right, then realize I need to add some more things to the `__init__`, and have to re-create the entire class again, and then again, and again, and oh, the regrets I will feel for having reached for the wrong tool out of my old, hard-to-shed habits. If I'd been working with the Jupyter Notebook, I'd just change the error directly in the cell containing the code, without any up-arrow shenanigans, and be on my way in seconds (see Figure 1-1).


jupyter Untitled (autosaved)
File Edit View Insert Cell Kernel Help
Python 2
Markdown
Cell Toolbar: None

Hello world

We can make mistakes in Jupyter Notebook just as easily as we can in a plain old Python shell, but it's a lot faster to *fix them* here!

```
In [5]: class Foo(object):

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

            def bar(self):
                retrun self.x

          File "<ipython-input-5-9e4aa4e1533c>", line 7
            retrun self.x
                      ^
        SyntaxError: invalid syntax
```

```
Cells can contain code or Markdown.
Notebooks can be rapidly-prototyped QA tools,
the basis of scientific papers,
and even power slideshows for your
next conference speaking gig!
```

```
In [ ]:
```

Figure 1-1. The Jupyter Notebook gives your browser super powers!

It takes just a little bit of extra effort and forethought to install and learn your way around one of these more sophisticated REPLs, but the sooner you do, the happier you'll be.

CHAPTER 2

Silly Things

Oops! I did it again.
—Britney Spears

There's a whole category of just plain silly mistakes, unrelated to poor choices or good intentions gone wrong, the kind of strangely simple things that I do over and over again, usually without even being aware of it. These are the mistakes that burn time, that have me chasing problems up and down my code before I realize my trivial yet exasperating folly, the sorts of things that I wish I'd thought to check for an hour ago. In this chapter, we'll look at the three silly errors that I commit most frequently.

Forgetting to Return a Value

I'm fairly certain that a majority of my hours spent debugging mysterious problems were due to this one simple mistake: forgetting to return a value from a function. Without an explicit `return`, Python generously supplies a result of `None`. This is fine, and beautiful, and Pythonic, but it's also one of my chief sources of professional embarrassment. This usually happens when I'm moving too fast (and probably being lazy about writing tests)—I focus so much on getting to the answer that returning it somehow slips my mind.

I'm primarily a web guy, and when I make this mistake, it's usually deep down in the stack, in the dark alleyways of the layer of code that shovels data into and out of the database. It's easy to get distracted by crafting just the right join, making sure to use the best indexes, getting the database query *just so*, because that's the *fun* part.

Here's an example fresh from a recent side project where I did this yet again. This function does all the hard work of querying for voters, optionally restricting the results to voters who cast ballots in some date range:

```
def get_recent_voters(self, start_date=None, end_date=None):
    query = self.session.query(Voter).\
            join(Ballot).\
            filter(Voter.status.in_(['A', 'P']))
    if start_date:
        query.filter(Ballot.election_date >= start_date)
    if end_date:
        query.filter(Ballot.election_date <= end_date)
    query.group_by(Voter.id)
    voters = query.all()
```

Meanwhile, three or four levels up the stack, some code that was expecting to iterate over a list of `Voter` objects vomits catastrophically when it gets a `None` instead. Now, if I've been good about writing tests, and I've only just written this function, I find out about this error right away, and fixing it is fairly painless. But if I've been In The Zone for several hours, or it's been a day or two between writing the function and getting a chance to exercise it, then the resulting `AttributeError` or `TypeError` can be quite baffling. I might have made that mistake hundreds or even thousands of lines ago, and now there's so much of it that looks correct. My brain knows what it *meant* to write, and that can prevent me from finding the error as quickly as I'd like.

This can be even worse when the function is expected to sometimes return a `None`, or if its result is tested for truthiness. In this case, we don't even get one of those confusing exceptions; instead the logic just doesn't work quite right, or the calling code behaves as if there were no results, even though we know there should be. Debugging these cases can be exquisitely painful and time-consuming, and there's a strong risk that these errors might not be caught until much later in the life cycle of the code.

I've started to combat this tendency by cultivating the habit of writing the `return` immediately after defining the function, making a second pass to write its core behavior:

```
def get_recent_voters(self, start_date=None, end_date=None):
    voters = []
    # TODO: go get the data, sillycakes
    return voters
```

Yes, I like to sass myself in comments; it motivates me to turn `TODO` items into working code so that no one has to read my twisted inner monologue.

Misspellings

One of the top entries on my list of superpowers is my uncanny ability to mistype variable or function names when I'm programming. Like my forgetfulness about returning things from functions, I encounter this the most when I've been In The Zone for a couple of hours and have been slacking at writing or running tests along the way. There's nothing quite like a pile of `NameErrors` and `AttributeErrors` to deflate one's ego at the end of what seemed like a glorious triumph of programming excellence.

Transposition is especially vexing because it's hard to see what I've done wrong. I know what it's *supposed* to say, so that's all I can see. Worse, if the flaw isn't exposed by tests, there's a good chance it will escape unscathed from code review. Peers reviewing code can skip right over it because they also know what I'm getting at and assume (often too generously) I know what I'm doing.

My fingers seem to have certain favorites that they like to torment me with. Any end-to-end tests I write against our REST APIs aren't complete without at least half a dozen instances of `respones` when I mean `response`. I may want to add a `metadata` element to a JSON payload, but if it's getting close to lunch time, my rebellious phalanges invariably substitute `meatdata`. Some days I just give in and deliberately use `slef` everywhere instead of `self` since it seems like my fingers won't cooperate anyway.

Misspelling is particularly maddening when it occurs in a variable assignment inside a conditional block like an `if`:

```
def fizzbuzz(number):
    output = str(number)
    if number % 3 == 0:
        putput = "fizz"
    ...
    return output
```

The code doesn't blow up, no exceptions are raised—it just doesn't work right, and it is utterly exasperating to debug.

This issue, of course, is largely attributable to my old-school, artisinal coding environment, by which I mean I've been too lazy to invest in a proper editor with auto-completion. On the other hand, I've gotten good at typing `xp` in Vim to fix transposed characters.

I have also been really late to the Pylint party. Pylint is a code analysis tool that examines your code for various "bad smells." It will warn you about quite a lot of potential problems, can be tuned to your needs (by default, it is rather talkative, and its output should be taken with a grain of salt), and it will even assign a numeric score based on the severity and number of its complaints, so you can gamify improving your code. Pylint would definitely squawk about undefined variables (like when I try to examine `respones.headers`) and unused variables (like when I accidentally assign to `putput` instead of `output`), so it's going to save you time on these silly bug hunts even though it may bruise your ego.

So, a few suggestions:

- Pick an editor that supports auto-completion, and use it.
- Write tests early and run them frequently.
- Use Pylint. It will hurt your feelings, but that is its job.

Mixing Up Def and Class

Sometimes I'm working head-down, hammering away at some code for a couple of hours, deep in a trance-like flow state, blasting out class after class like nobody's business. A few hundred lines might have emerged from my fingertips since my last conscious thought, and I am ready to run the tests that prove the worth and wonderment of my mighty deeds.

And then I'm baffled when something like this…

```
class SuperAmazingClass(object):

    def __init__(self, arg1, arg2):
        self.attr1 = arg1
        self.attr2 = arg2

    def be_excellent(to_whom='each other'):
        ...

    # many more lines...


def test_being_excellent():
    instance = SuperAmazingClass(42, 2112)
    assert instance.be_excellent(...)
```

…throws a traceback like this:

```
    TypeError:  SuperAmazingClass()  takes  exactly  1  argument  (2
    given)
```

Wait, *what?*

My reverie is over, my flow is gone, and now I have to sort out what I've done to myself, which can take a couple of minutes when I've been startled by something that I assumed should Just Work.

When this happens, it means that I only *thought* that I wrote the code above. Instead, my careless muscle memory has betrayed me, and I've *really* written this:

```
    def SuperAmazingClass(object):

        def __init__(self, arg1, arg2):
            ...
```

Python is perfectly content to define functions within other functions; this is, after all, how we can have fun toys like closures (where we return a "customized" function that remembers its enclosing scope). But it also means that it won't bark at us when we mean to write a class but end up accidentally definining a set of nested functions.

The error is even more confusing if the `__init__` has just one argument. Instead of the `TypeError`, we end up with:

```
    AttributeError:  'NoneType'  object  has  no  attribute  'be_excel-
    lent'
```

In this case, our "class" was called just fine, did nothing of value, and implicitly returned `None`. It may seem obvious in this contrived context, but in the thick of debugging reams of production code, it can be just plain *weird*.

Above all, be on your guard. Trust no one—least of all yourself!

CHAPTER 3
Style

Okay, so ten out of ten for style, but minus several million for good thinking, yeah?
—Zaphod Beeblebrox

In this chapter, we're going to take a look at five ways I've hurt myself with bad style. These are the sorts of things that can seem like a good idea at the time, but will make code hard to read and hard to maintain. They don't break your programs, but they damage your ability to work on them.

Hungarian Notation

A great way to lie to yourself about the quality of your code is to use Hungarian Notation. This is where you prefix each variable name with a little bit of text to indicate what kind of thing it's supposed to be. Like many terrible decisions, it can start out innocently enough:

```
strFirstName
intYear
blnSignedIn
fltTaxRate
lstProducts
dctParams
```

Or perhaps we read part of PEP-8 and decided to use underscores instead, or like suffixes more than prefixes. We could make variables like these:

```
str_first_name
products_list
```

The intent here is noble: we're going to leave a signpost for our future selves or other developers to indicate our intent. Is it a string? Put a `str` on it. An integer? Give it an `int`. Masters of brevity that we are, we can even specify lists (`lst`) and dictionaries (`dct`).

But soon things start to get silly as we work with more complex values. We might conjoin `lst` and `dct` to represent a list of dictionaries:

```
lctResults
```

When we instantiate a class, we have an object, so `obj` seems legit:

```
objMyReallyLongName
```

But that's an awfully long name, so as long as we're throwing out unneeded characters, why not boost our job security by trimming that name down even further, to the point that it's completely meaningless:

```
objMRLN
```

Maybe we don't know what kind of data we're going to have:

```
varValue
```

Before long, we're straight-up lying, creating variables like these—a number that isn't a number, and a boolean that isn't a boolean:

```
strCustomerNumber = "123456789"
blnFoo = "N"
```

This, in turn, is a gateway to logic either silently failing (a "boolean" that's actually a string will always be "truthy" in an `if` or `while`) or throwing mysterious `AttributeError` exceptions that can be particularly difficult to diagnose if you have several of these liars in a single expression (such as when you're formatting a string, and one of them is accidentally a `None` in disguise). It also limits our thinking: when we read `products_list` or `lstResults`, we won't ever expect that they might be generators or some other kind of sequence. Our thoughts are tied to specific types, when we might be better served by thinking at a higher level of abstraction.

At the best, we make everything a few characters longer and harder to read; at the worst, we lie to ourselves and introduce frustrating runtime errors. So when it comes to Hungarian Notation, just say `blnNo`!

PEP-8 Violations

When I was starting out in Python, I picked up some bad habits from our existing codebase and perpetuated them for a lot longer than I should have. Several years had passed before I discovered PEP-8, which suggests a standardized style for writing Python code. Let's take a look at a distilled example and examine my sins:

```
class MyGiganticUglyClass(object):
  def iUsedToWriteJava(self,x,y = 42): ❶ ❷ ❸
    blnTwoSpacesAreMoreEfficient = 1 ❶ ❸ ❹
    while author.tragicallyConfused(): ❸
       print "Three spaces FTW roflbbq!!1!" ❶
    if (new_addition): ❺
        four_spaces_are_best = True ❶
    if (multipleAuthors \ ❸ ❻
        or peopleDisagree): ❸
            print "tabs! spaces are so mainstream" ❶
    ...
    return ((pain) and (suffering)) ❺
```

❶ **Indentation issues:** At first I thought three spaces of indentation were pretty great, then I realized that two spaces meant I could pack more code onto a line. Sometimes that'd be mixed with tabs, while newer, more enlightened additions would use the recommended four spaces. Mixing tabs and spaces can be especially dangerous as well, as it can cause logic to fail in interesting and unexpected ways at runtime, usually at night, always when you're on call. Just because it *looks* like a line is indented properly to be within a particular block doesn't mean it actually *is* if tabs are involved!

❷ **Whitespace issues:** I've omitted whitespace after commas, yet added unnecessary whitespace around keyword arguments. And the whole thing would be more readable with some blank lines in between the class and function definition, as well as within the function, to better separate ideas.

❸ **Inconsistent case:** While `mixedCaseNames` might be the standard practice for functions and variables in other languages (Java, JavaScript), the Python community prefers `lowercase_with_underscores` for enhanced readability.

❹ **Hungarian notation:** 'nuff said.

❺ **Extraneous parentheses:** Parentheses are optional around expressions, and in some cases may improve readability or help clarify order of operations, but in this case we don't need them when checking a single value in an `if` block or in the way-too-complicated `return` statement.

❻ **Extraneous line continuations:** If you're inside a set of parentheses (such as when calling a function or defining a generator expression), square brackets (defining a list or list comprehension), or curly braces (defining a set, dictionary, or comprehension), you don't need the backslash to continue the statement on the next line.

If we tidied this up a little, it might look better like this:

```
class MyMorePythonicClass(object):

    def now_i_write_python(self, x, y=42):
        two_spaces_hamper_readability = True

        while author.tragically_confused(): ❶
            print "Three spaces? What was I thinking?"

        if new_addition:
            four_spaces_are_best = True

        if (multiple_authors or
                people_disagree): ❷
            print "Mixing tabs and spaces is dangerous!"

        ...

        return sunshine and puppies
```

❶ At first it might seem like there's too much whitespace within this method, but once a given block has more than three or four lines, or there are more than two or three blocks, it's really much more readable.

❷ This is actually short enough to have the whole expression on one line with no continuation and no parentheses, but then it wouldn't illustrate this improved multiline style.

There's also an appealing foolishness to getting everything lined up just right:

```
from regrets            import unfortunate_choices

class AnotherBadHabit(object):

    short_name          = 'foo'
    much_longer_name    = 'bar'

    def __init__(self,  x, y, z):
        self.x = x
        self.y = y
        self.z_is_a_silly_name = z
        self.came_later        = 42
        self.mild_disappointment = True
        self.leftover    = 'timewaster'
        self.dictionary = {
            'foo'       : 'bar',
            'bar'       : 'baz',
            'baz'       : 'quux',
        }
```

I did this a *lot* in my initial years with Python, I think as a reaction to existing code that I didn't consider well-formatted or terribly readable. It seems pleasing at first, but before long, something needs to be added, or changed, and you're quickly locked into a spiral of despair and spend all your time adjusting the internal whitespace of each line of code. Inevitably, you're stuck with weird artifacts and inconsistencies that will haunt your nightmares as well as make lots of unnecessary noise in your diffs and code reviews.

Don't do this—it will just drive you crazy.

Bad Naming

At some point I internalized PEP-8's 80-character line length limit, but my poor judgment led me to squeeze the most code I could into a single line by using single-character variables wherever possible:

```
f.write(string.join(map(lambda
x,y=self.__dicProfiles,z=strPy:"%0.3s %s:
%s:(%s)" % (z,x,y[x][0],y[x]
%[1]),self.__dicProfiles.keys()),'\n')
%+'\n')
```

Such meaningless variable names lead to code that's really hard to read, and people are afraid to clean it up. I have no idea what this even does anymore!

Single-character variable names are also awful to search for when debugging or trying to make sense of code. Imagine asking your editor to show you every s or n in a large block of text; it will be nearly impossible to find what you want in a sea of false positives.

And since callables are first-class citizens in Python, we can produce nightmares like this by assigning functions to single-letter (and extremely short) variables, too:

```
#!/usr/bin/env python
import os,sys
C=os.chdir
S=os.system
M=os.mkdir
J=os.path.join
A=os.path.abspath
D=os.path.dirname
E=os.path.exists
W=sys.stdout.write
V=sys.argv
X=sys.exit
ERR=lambda m:W(m+"\n")
PRNT=lambda m:W(m+"\n")
assert len(V)==2,"you must provide a name"
SB=V[1]
H=A(D(__file__))
SBD=J(D(H),SB)
C(SBD)
...
X(0)
```

Stare deeply into a line of code like SBD=J(D(H),SB) and it's like gazing into the abyss. The cognitive load of deciphering this later simply isn't worth it—give things meaningful, human-readable names.

Of course, it's entirely possible to hurt yourself with long names, too. If you aren't working with an editor that can do auto-completion, things like these are filled with peril:

```
class TestImagineAClassNameThatExceeds80Characters(object):
    ...

def getSomethingFancyfromDictionary(...):
    ...

count_number_of_platypus_incidents_in_avg_season = ...
```

Will you remember the right spellings or capitalization? (Was it "number" or "num"? "Average" or "avg"? "From" or "from"?) Will you spot the typos? Will you even be able to read the code that uses these names?

foo, bar, and baz are a good fit for example code, but not something that has to run and be maintained in production. The same goes for every silly, nonsense name you might be tempted to use. Will you even remember what spam or moo do in a week? In six months? I once witnessed classes named for post-Roman Germanic tribes. Pop quiz: What does a Visigoth do? How about a Vandal? These names might as well have been line noise for all the good they did.

Though it grieves me to say it, clever or nerdy cultural references (my worst offenses were lois.py and clark.py, which did some reporting tasks, and threepio.py, which communicated with a partner's "EWOKS" system) should be avoided as well. Inevitably, you will be devastated when no one appreciates the joke. Save the comedy for your code comments.

Even semantically accurate but cute names can be a source of pain. You'll command a lot more self-respect when you opt for LocationResolver over LocationLookerUpper.

Names should be clear, concise, specific, meaningful, and readable. For a great exploration of this topic, check out Brandon Rhodes' talk from PyCon 2013, "The Naming of Ducks".

Inscrutable Lambdas

You can create anonymous functions inline in your Python code with lambdas. Using lambdas can make you feel really smart, but I've become progressively allergic to them. Even when they're simple, they can be hard to read and quickly become confusing if there's more than one on a line or in an expression:

```
lstRollout = filter(lambda x: x[-1] == '0',
        filter(lambda x: x != '0|0', lstMbrSrcCombo))

if not filter(lambda lst, sm=sm: sm in lst,
        map(lambda x, dicA=dicA: dicA.get(x, []),
            lstAttribute)):
    ...
```

When we use a lambda in the middle of a line of code, that 80-character rule pressures us to really make the most of that line. Cue the one- and two-character variable names!

```
_make_keys = lambda cc, p: tuple(map(
        lambda m, c=cc: ("%s.%s" % (c, m), m), p))
```

Because the functions created by the lambda are anonymous, we can't give them meaningful names that would express what's going on. Every time we read them, we have to figure them out all over again.

These anonymous functions are also not reusable, which means that if we're repeatedly using them for the same purpose, we stand a much larger chance of screwing one of them up. If we're lucky, it breaks in a way that gives us an exception to chase down. Otherwise, we've got a very subtle bug that's hard to pinpoint because it's hard to see the error in mostly alike code:

```
foo = map(lambda x: x[-1].replace('taco', 'cat'), foos)
bar = map(lambda x: x[-1].replace('tacp', 'cat'), bars)
baz = map(lambda x: x[-1].replace('taco', 'cat'), bazzes)
```

Our future selves will often be better off if we extract that complexity into a named, reusable, documentable, testable function that we only have to get right once:

```
def taco_to_cat(input):
    """Convert tacos to cats"""
    return input[-1].lower().replace('taco', 'cat')


foo = map(taco_to_cat, foos)
bar = map(taco_to_cat, bars)
baz = map(taco_to_cat, bazzes)
```

Incomprehensible Comprehensions

List comprehensions are great: they're beautiful, they're elegant, they're inspiring other languages to adopt them. When I discovered list comprehensions, I fell in love, and I fell *hard*. I used them at every opportunity I had. And using them is fine, until they get filled with so much junk that it's hard to see what's even going on.

This example isn't *too* bad, but any time comprehensions are nested like this, it takes more effort to understand what's happening:

```
crumbs = [y for y in
        [x.replace('"', '') for x in crumbs] if y]
```

This one will scare new developers who aren't friends with `zip` yet:

```
return [dict(x) for x in [zip(keys, x) for x in values]]
```

And this one's just freaky:

```
prop_list = [
        FilterProp(prop='P_EXCLUDED', data='_'.join([i, j, k]))
        for i in prop_data[0]
        for j in prop_data[1]
        for k in prop_data[2]]
```

All of those examples are real, all of them appeared inline in other functions, and none of them were commented or explained. (I am so, so sorry.) At the very least, constructions like this deserve some kind of comment. They could probably use better variable names than `x` or `j` (and in the `i`, `j`, `k` case, those weren't even integers for counting—oof!).

If the comprehension is sufficiently complex, it might even be worth extracting the whole thing into a separate function with a reasonable name to encapsulate that complexity. Instead of the examples above, imagine if we had code that read like this:

```
crumbs = filter_crumbs(crumbs)

data = dict_from_lists(keys, values)

prop_list = make_exclusion_properties(prop_data)
```

There might still be complexity lurking behind those function calls, but it's all got a chance to have a name, a docstring, and unit tests that can validate it.

NOTE

Though we focused on list comprehensions here, the same perils and possibilities apply to dictionary and set comprehensions as well. Use them wisely, and only for good.

CHAPTER 4
Structure

It's a trap!
—Admiral Ackbar

Let's move on into questions of structure and how you can hurt your future self with tangled logic and deep coupling. These structural problems impact your ability to change or reuse your code as its requirements inevitably change.

Pathological If/Elif Blocks

This anti-pattern arises when you get into the business of creating a "one-stop shop" function that has to contend with many special cases.

The first `if/else` block arrives innocently, and even the first `elif` doesn't seem so bad. But soon their friends arrive:

```python
def do_awesome_stuff():
    ...
    if thing.has_condition_one():
        ...
    elif thing.has_condition_two():
        ...
    elif thing.get_conditions() in ['conditon3', 'condition4']:
        ...
    elif thing.has_condition_forty_two():
        ...
    else:
        ...
    ...
```

Suddenly you find yourself with hundreds of lines of elifs. And good luck if any of the contents of those blocks is at all complicated —anyone reading this code will be fortunate if they even remember they're in this elif nightmare after 30 or 40 lines. And how excited will you be to write tests for this function?

This has a kind of momentum as well—special cases tend to attract more special cases, as if drawn together gravitationally. Just adding more elifs feels easier than cleaning up. Except cleaning up isn't so bad. If we really do need to manage many special cases, we can employ the Strategy pattern:

```
def strategy1():
    ...

def strategy2():
    ...

strategies = {
    'condition1': strategy1,
    'condition2': strategy2,
    ...
}

def do_awesome_stuff():
    which_one = ...
    strategy = strategies[which_one]
    strategy()
    ...
```

We start by extracting the contents of our if/elif/else structure into separate functions with identical interfaces. Then we can create a dictionary to map conditions to those strategy functions. The dictionary key doesn't have to be a string. It can be anything hashable, so tuples and frozensets can be quite effective if we need richer conditions. Finally, our original function determines which key to use, plucks the appropriate strategy function from our dictionary, and invokes it.

Our original function is now much, much simpler to understand, as are each of the strategies, and writing tests for each of the now-isolated strategies is straightforward.

However, figuring out what value to use for that dictionary key can sometimes be complicated. If it takes 200 lines to determine what key to use, is this really much of a victory?

If that's the case, consider externalizing it entirely, and let the strategy be chosen by the caller, who may in fact know better than we do about whatever those factors are. The strategy is invoked as a callback:

```
def do_awesome_stuff(strategy):
    ...
    strategy()
    ...


result = do_awesome_stuff(strategy1)
```

From there it's not too far of a jump into dependency injection, where our code is provided with what it needs, rather than having to be smart enough to ask for it on its own:

```
class Foo(object):

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

    def do_awesome_stuff(self):
        ...
        self.strategy()
        ...


foo = Foo(strategy2)
foo.do_awesome_stuff()
```

Unnecessary Getters and Setters

In between Perl and Python, there was a brief window where I was immersed in Java, but its influence lingered far beyond those few months. When I got to do some of my first brand new, greenfield development of an invitation service, I made sure that all of the model objects were replete with getters and setters because, darn it, this was how object-oriented programming was supposed to be! I would show them all—attribute access must be protected!

And thus it was that I produced many classes that looked like this:

```
class InviteEvent(object):
    ...

    def getEventNumber(self):
        return self._intEventNumber
    def setEventNumber(self, x):
        self._intEventNumber = int(x)

    ...
```

Each and every attribute of each and every class had getter and setter functions that did barely anything. The getters would simply return the attributes that they guarded, and the setters would occasionally enforce things like types or constraints on the values the attributes were allowed to take. This `InviteEvent` class had 40 getters and 40 setters; other classes had even more. That's a lot of code to accomplish very little—and that's not even counting the tests needed to cover it all.

And trying to work with instances of these objects was pretty awful, too—this kind of thing quickly becomes tiresome:

```
event.setEventNumber(10)
print event.getEventNumber()
```

Fortunately, there's a practical, Pythonic solution to this labyrinth of boilerplate: just make most attributes public, and use properties to protect any special snowflakes that need extra care and feeding.

Properties let you provide functions that masquerade as attributes of the object: when you read the attribute, a getter is invoked; when you assign to the attribute, a setter is called; when you try to delete the attribute, it's managed by a deleter. The setter and deleter are both optional—you can make a read-only attribute by declaring only the getter. And the really great thing is that you don't need to know in advance which attributes will need to be properties. You have the freedom to sketch out exactly what you want to work with, then transparently replace attributes with properties without having to change any calling code because the interface is preserved.

In modern Python, properties are constructed with the `@property` decorator, which is just syntactic sugar for a function that replaces a method with a property object of the same name and wires it up to the getter. The property object also has `setter` and `deleter` functions that can be used as decorators to attach setter and deleter functionality to the property.

That might sound complicated, but it's actually rather clean:

```
class InviteEvent(object):
    ...
    @property
    def event_number(self):
        return self._event_number

    @event_number.setter
    def _set_event_number(self, x):
        self._event_number = int(x)

    @event_number.deleter
    def _delete_event_number(self):
        self._event_number = None

    ...
```

The only trick is remembering to use the name of the property when hooking up the setter or deleter, rather than using `@property` itself.

One nice thing about this decorator-based approach is that it doesn't junk up the namespace of the class with a bunch of functions that you really don't want anyone to call. There's just the single property object for each property!

Using these objects is far more comfortable than before, too. All those function calls and parentheses simply vanish, leaving us with what looks like plain old "dot" access:

```
event.event_number = 10
print event.event_number
```

Getting Wrapped Up in Decorators

One of the things I was most excited about as Python evolved was the opportunity to use decorators to attach reusable functionality to functions and methods. We saw its benefits above with `@property`.

A decorator is a function (or, more generally, a callable) that returns a function, which replaces the function being decorated. Imagine a small nesting doll (the function being decorated), placed inside another nesting doll (the "wrapper" function returned by the decorator). We use the syntactic sugar of the `@` symbol to apply decorators to functions being decorated.

Here's a simple decorator that wraps a function in another function that does something special before allowing the first function to be executed:

```
def my_decorator(function):
    def wrapper(*args, **kwargs):
        # do something special first
        ...
        return function(*args, **kwargs)
    return wrapper

@my_decorator
def foo(x, y, z):
    ...
```

Typical uses for decorators involve altering or validating the input to a function, altering the output of a function, logging the usage or timing of a function, and—especially in web application frameworks—controlling access to a function. You can apply as many decorators as you want, too—it's nesting dolls all the way down!

Decorators sound pretty swell, so why are we talking about them in a book about mistakes?

When you use Python's decorator syntax to wrap and replace functions, you immediately couple the original function to all the behavior that comes with the wrapper. If the original function is about making some calculation and the wrapper is about logging, the result is a function that's inescapably, inextricably about *both* of those concerns. This coupling is compounded with each additional decorator that's applied.

Did you want to test the original function in isolation? Too bad—that function is effectively *gone*. Your test has no choice but to exercise the final, multilayered Frankenstein function, which means you may have a series of unpleasant hoops to jump through in order to set up the test, none of which is material to the problem the original function is attempting to solve. The same goes for trying to call that original function in your production code—once the decorators have been applied, you're stuck with all the extra baggage that comes with them.

As a web developer, I encounter this the most when writing unit tests for controller methods ("views" in the Django parlance), because I often have several layers applied. A typical example might look something like this:

```
class MyController(object):

    @require_https
    @require_signed_in
    @validate_form(SomeForm(), ...)
    @need_database_connection
    def handle_post(self, request):
        ...
        return HTTPResponse(...)
```

It can be hugely beneficial to have those access controls written in a way that they can quickly be reused throughout the application, but it means that if I'm going to write tests, I have to do all the work required to fake out the request context so that the request will actually make it to the code that I want to test. In an ideal world, the innermost method I'm testing is simple and doesn't need more than one or two tests to cover its behavior, but if it's at all complicated, the amount of setup necessary can become quite tedious (unless of course you get excited about refactoring unit tests, in which case have at it!).

And all of that setup means that I'm not only testing the original function, but in effect I'm testing all of the wrappers that the function has been decorated with, each of which should already have tests of their own.

The approach I've gravitated toward is to make the decorated method as simple and devoid of logic as possible, pushing all of its smarts down into a deeper layer of abstraction that can be tested in isolation:

```
class MyController(object):

    @require_https
    @require_signed_in
    @validate_form(SomeForm(), ...)
    @need_database_connection
    def handle_post(self, request):
        # get data from request
        data = { ... }
        self.object_service.create_object(data)
        return HTTPResponse(...)
```

Then the responsibility of the controller method is limited to receiving the request and handing the right data off to someone else, which makes its tests simpler as well. It also means that the core business logic is relocated away from the web interface and into a position that allows it to be reused.

Breaking the Law of Demeter

The Law of Demeter (also known as the principle of least knowledge) tells us that our code should only interact with the things that it knows about, and not reach deeply into nested attributes, across friends of friends, and into strangers.

It feels great to break this law because it's so expedient to do so. It's easy to feel like a superhero or a ninja commando when you quickly tunnel through three, four, or more layers of abstraction to accomplish your mission in record time.

Here are just a few examples of my countless crimes. I've reached across multiple objects to call a method:

```
gvars.objSession.objCustomer.objMemberStatus.isPAID()
```

Or reached through dictionaries to call a method to get an object to use to call another method:

```
if gvars.dctEnv['session'].getCustomer().isSignedIn():
        ...
```

Or called single-underscore-prefixed internal methods of an object: (more on this in a moment):

```
current_url = self.objSession._getCurrentURL()
```

Or called a method on an item plucked from a list returned by a method call on a single-underscore internal attribute of an object:

```
return event._objGuestList.getGuestList()[0].getEventSequence()
```

Yikes!

This kind of thing might be okay when we're debugging, or exploring in an interactive shell, but it's bad news in production code. When we break this law, our code becomes brittle. Instead of relying on the public interface of a single object, it now relies on a delicate chain of nested attributes, and any change that disrupts that chain will break our code in ways that will furrow our brows as we struggle to repair the complex code plumbing mess we've made for ourselves.

We should especially avoid depending on single- and double-underscore internals of an object, because they are prefixed this way for a reason. We are explicitly being told that these items are part of the internal implementation of the object and we cannot depend on them to remain as they are—they can be changed or removed at any time. (The single underscore is a common convention to indicate that whatever it prefixes is "private-ish," while double-underscore attributes are made "private" by Python's name mangling.)

The problem of these violations is even worse than it seems, for it turns out that the brittleness and calcification of the system happens in both directions. Not only is the calling code locked into the internal interfaces that it's traversing, but each and every object along that path becomes locked in place as well, as if encased in amber. None of these objects can be freely or easily changed, because they are all now tightly coupled to one another.

If it really is the responsibility of an object to surface something from deep within its internals, make that a part of the object's public interface, a first-class citizen for calling code to interact with. Or perhaps an intermediary helper object can encapsulate the traversal of all those layers of abstraction, so that any brittleness is isolated to a single location that's easy to change instead of woven throughout the system. Either way, let abstraction work *for* you. This frees both the caller and callee to change their implementations without disrupting each other, or worse, the entire system.

Overusing Private Attributes

When I started with Python, I was still fresh out of school, where I'd heard over and over again about the importance of object-oriented programming ideals like "information hiding" and private variables. So when I came to Python, I went a little overboard with private methods and attributes, placing leading double underscores on practically everything I could get my hands on:

```
class MyClass(object):

    def __init__(self, arg1, arg2, ...):
        self.__attr1 = arg1
        self.__attr2 = arg2
        ...

    def do_something(self):
        self.__do_a_step()
        self.__do_another_step()
        self.__do_one_more_step()
        self.__do_something_barely_related()

    # and so forth...
```

"Hands off!" this code shouts. "You'll never need to use these things, and I know better than you!"

Inevitably, I discovered that I *did* need to use code that was hiding behind the double underscore, sometimes to reuse functionality in previously unforeseen ways, sometimes to write tests (either to test a method in isolation or to mock it out).

Let's say we wanted to subclass that `MyClass` up above, and it needs a slightly customized implementation of the `do_something` method. We might try this:

```python
class MyOtherClass(object):

    def do_something(self):
        self.__do_a_new_step()
        self.__do_one_more_step()
```

This will fail with an `AttributeError`, because the name mangling that Python applies to make the attribute private means that our subclass won't actually *have* a `__do_one_more_step` method to call. Instead, we would have to invoke `self._MyClass__do_one_more_step`, and that's just nasty.

All that privacy just got in the way. What at first seemed like a cool language feature turned out to be a giant nuisance that I was always working around.

In time, I came to prefer using just a single underscore to politely indicate that an attribute is part of the internal implementation of a class and shouldn't be referenced externally without some hesitation. Since single-underscore names aren't mangled, they can be more conveniently used if you absolutely must break the Law of Demeter.

Further experience taught me that I shouldn't even want to do that. When "outside" code wants access to the internals of a class, those "internal" attributes probably shouldn't be private at all; rather, this is a clear signal that those attributes should be public. The code is telling us that it's time to refactor!

The "private" attribute we keep using externally should either be promoted to not have any leading underscores, or should be exposed via a property if some amount of control is still required. If we feel the need to replace or monkey-patch an internal method of a class, we should instead be thinking about extracting that into a strategy that perhaps we just pass in to the public method we're calling. If we find that we need to call into that "internal" functionality in multiple places, then what the code is telling us is that that functionality doesn't really belong in this class at all. It should be extracted into a separate function, or if complex enough, into a separate object that the class might collaborate with rather than wholly contain.

God Objects and God Methods

A well-behaved class or method should have a strictly limited set of responsibilities, preferably as close to one as possible (in accordance with the Single Responsibility Principle), and should only contain whatever knowledge or data it needs to fulfill its limited role. All classes and methods start this way, simple and innocent, but we may find it convenient or expedient to grow these entities as our requirements evolve. When a class or method has accumulated too much knowledge or too many responsibilities, its role in the system becomes practically godlike: it has become all-encompassing, all-seeing, and all-doing, and many other entities will end up being tightly coupled to it in order to get anything done. Like big banks in the autumn of 2008, our god objects and god methods are too big to maintain, yet too big to fail.

These are pretty easy to spot: we're looking for large modules, large classes, classes with many methods, and long methods or functions. There are a couple of different ways to go about this.

Pylint will by default complain about modules longer than 1000 lines and functions longer than 50 lines (and you can adjust these values as needed), but you have to look carefully at its voluminous output to make sure you don't miss these warnings. WingIDE and Komodo integrate with Pylint for code inspection, so they'll also help you find these problems. Curiously, while PyCharm offers code inspection that covers many of the same issues that Pylint does, it doesn't include warnings about module or function length.

If you aren't someone who enjoys working with an IDE, you can use some Unix command-line kung fu to identify potential sources of godlike trouble:

```
$ find . -name "*.py" -exec wc -l {} \; | sort -r ❶
$ grep "^class " bigmodule.py | wc -l ❷
$ grep "\sdef " bigmodule.py | wc -l ❸
```

❶ Find all Python soure files, count the number of lines, and sort the results in descending order, so that the files with the most lines bubble to the top of the list; anything over 1000 lines is worth further investigation.

❷ Count the number of classes defined in a big module…

❸ And the number of methods defined at some level of indentation (i.e., within a class or within other functions) in that module.

If the ratio of methods to classes seems large, that's a good warning sign that we need to take a closer look.

Or, if we feel like being creative, we can use Python to make a little cross-platform tool:

```python
import collections
import fileinput
import os


def find_files(path='.', ext='.py'):
    for root, dirs, filenames in os.walk(path):
        for filename in filenames:
            if filename.endswith(ext):
                yield(os.path.join(root, filename))


def is_line(line):
    return True


def has_class(line):
    return line.startswith('class')


def has_function(line):
    return 'def ' in line


COUNTERS = dict(lines=is_line, classes=has_class,
        functions=has_function)


def find_gods():
    stats = collections.defaultdict(collections.Counter)
    for line in fileinput.input(find_files()):
        for key, func in COUNTERS.items():
            if func(line):
                stats[key][fileinput.filename()] += 1
    for filename, lines in stats['lines'].most_common():
        classes = stats['classes'][filename]
        functions = stats['functions'][filename]
        try:
            ratio = "=> {0}:1".format(functions / classes)
        except ZeroDivisionError:
            ratio = "=> n/a"
        print filename, lines, functions, classes, ratio


if __name__ == '__main__':
    find_gods()
```

This small program is enough to recursively find all `.py` files; count the number of lines, classes, and functions in each file; and emit those statistics grouped by filename and sorted by the number of lines in the file, along with a ratio of functions to classes. It's not perfect, but it's certainly useful for identifying risky modules!

Let's take a high-level look at some of the gods I've regretted creating over the years. I can't share the full source code, but their summaries should illustrate the problem.

One of them is called `CardOrderPage`, which spreads 2900 lines of pain and suffering across 69 methods, with an 85-line `__init__` and numerous methods in excess of 200 to 300 lines, all just to shovel some data around.

`MemberOrderPage` is only 2400 lines long, but it still packs a whopping 58 methods, and its `__init__` is 90 lines. Like `CardOrderPage`, it has a diverse set of methods, doing everything from request handling to placing an order and sending an email message (the last of which takes 120 lines, or roughly 5 percent of the class).

Then there's a thing called `Session`, which isn't really what most web frameworks would call a session (it doesn't manage session data on the server), but which instead provides context about the request, which is a polite way to say that it's a big bag of things that you can hurt yourself with. Lots of code in this codebase ended up being tightly coupled to `Session`, which presents its own set of problems that we'll explore further in a later section.

At the time that I captured the data about it, `Session` was only about 1000 lines, but it had 79 methods, most of which are small, save for a monstrous 180-line `__init__` laden with mine fields and side effects.

Besides line count, another way you can identify god methods is by looking for naming anti-patterns. Some of my most typical bad methods have been:

```
def update_everything(...):
    ...

def do_everything(...):
    ...

def go(...):
    ...
```

If you find these kinds of abominations in your code, it's a sign that it's time to take a deep breath and refactor them. Favor small functions and small classes that have as few responsibilities as possible, and strive to do as little work as possible in the `__init__` so that your classes are easy to instantiate, with no weird side effects, and your tests can be easy and lighweight. You want to break up these wanna-be gods before they get out of hand.

Increasing the number of small classes and methods may not optimize for raw execution speed, but it does optimize for maintenance over the long term and the overall sanity and well-being of the development team.

Global State

We come now to one of my greatest regrets. This module is called `gvars.py`, and it started simply as a favor to another developer who needed easy access to some objects and didn't want to pass them around everywhere, from way at the top of the stack to deep down in the guts:

```
dctEnv = None
objSession = None
objWebvars = None
objHeaders = None
objUserAgent = None
```

It's basically just a module that has some module-level global variables that would get repopulated by the app server with every request that would come in over the web. If you import it, you can talk to those globals, and you can do this at any level, from those lofty heights that first see the request, where it seems like a reasonable thing to want to do, all the way down to the darkest, most horrible depths of your business logic, data model, and scary places where this has no business being. It enables this sort of thing at every level of your system:

```
from col.web import gvars
...

if gvars.objSession.hasSomething():
    ...

if gvars.objWebvars.get('foo') == 'bar':
    ...

strUserName = \
    gvars.objSession.objCustomer.getName()
```

This is tremendously convenient when you're writing website code —you can get at anything important about the request at any point, no matter where you are. Poof! Magic!

But as soon as you need to do anything else, it all falls apart. Any kind of script, cron job, or backend system is doomed, because if it needs to use anything that has been tainted by `gvars`, well, too bad! Code that uses `gvars` is immediately tightly coupled to the context of a web request, where normally the app server would set up all of those heavyweight objects based on the request. But outside of the context of a request, we don't have an app server, we don't get all those objects made for free, and even if we did, they wouldn't make sense—what is a user agent or POST variable in the context of a cron job?

The only hope for using code that's bound to `gvars` outside of its native milieu is to do extensive faking, populating `gvars` manually with objects that are good enough to get by, and providing your own specially crafted return values when necessary. This is less fun than it sounds.

Let's consider an example of the madness that `gvars` begat. The `PermissionAdapter` is a class that would fetch question data for managing user opt-ins and opt-outs for various flavors of email. Naturally, for convenience, it depended on a pile of objects created by the app server and injected into `gvars` at request time, and it was so internally knotty that refactoring it to clean it up was frowned upon, lest we end up doing more harm than good. No, it didn't have unit tests, why do you ask?

```
from col.web import gvars

class PermissionAdapter(object):

    def __init__(self, ...):
        # uh-oh, this can't be good...
        dctEnv = gvars.dctEnv
        self._objWebvars = dctEnv["webvars"]
        self._objSession = dctEnv["session"]
        self._objHeaders = dctEnv["headers"]

    def getQuestions(...):
        site = self._objSession.getSiteGroup()
        data = self._someThingThatWants(site)
        ...
```

For its specific purpose, it got the job done, albeit with little flexibility.

One day, I had to figure out how to surface the permission questions for more than one sitegroup (a fancy internal term for a customer namespace). Without some serious refactoring, this just wasn't possible as-is.

So instead—and I am so, so sorry for this—I wrote a `PermissionFacade` wrapper around the `PermissionAdapter`, and its job was to fake out the necessary objects in `gvars` using Mock objects, instantiate a `PermissionAdapter`, then restore the original `gvars` before leaving the method:

```
class PermissionFacade(object):

    def __init__(self, ...):
        self.webvars = Mock()
        self.session = Mock()
        self.headers = Mock()

    def get_questions(sitegroup, ..., whatever):
        adapter = self.make_adapter(...)
        return adapter.getQuestions(...)

    def _make_adapter(self, sitegroup, ...):
        from col.web import gvars
        orig_gvars_env = gvars.dctEnv
        gvars.dctEnv = {
            'webvars': self.webvars,
            'session': self.session,
            'headers': self.headers,
        }
        self.session.getSource.return_value = source
        self.session.getSourceFamily.return_value = \
                source_family
        try:
            self.permission_adapter = PermissionAdapter(
                    sitegroup, ...)
            # ...and some other grotesque mock monkey
            # patching to fake out a request context...
        finally:
            gvars.dctEnv = orig_gvars_env
        return self.permission_adapter
```

Thank goodness we at least have `finally` to give us the opportunity to put the original values back into place. This makes sure that no matter what happens during the corresponding `try`, we'll put everything back where it was so that we can service other callers. Subsequent calls can ask for questions for a diffent sitegroup, and we can then combine the results further upstream.

But patching things into place like this in production code is a bad idea, because it's just too easy to screw up in a way that's weird or subtle. Even in the normal web server context, something like `gvars` can lead to safety issues and possible data leakage between requests unless it's carefully managed, as those module-level globals will persist as long as the process is running. We'll see this come back to haunt us in the next chapter.

Avoid global state as much as humanly possible. Resist its siren lure, reject its convenience, refuse the temptation, no matter how much your colleagues think they want it. In the long run, they'll thank you for not having to maintain such monstrosities.

CHAPTER 5
Surprises

If you do not expect the unexpected you will not find it,
for it is not to be reached by search or trail.
—Heraclitus

At last, we get to the really weird stuff, the things that go bump in the night and cause someone to get paged to solve them. These are some of the many ways that you can create little time bombs in your code, just waiting to surprise and delight you at some point in the future.

Importing Everything

PEP-8 recommends avoiding wildcard imports (`from some_module import *`), and it's absolutely right. One of the most exciting reasons is that it opens your code up to some interesting ways to break in a multideveloper environment.

Suppose there's some module `foo.py` that has a bunch of great things in it, and your code wants to make use of many of them. To save yourself the tedium of either importing lots of individual names, or importing just the module and having to type its name over and over again, you decide to import everything:

```
import time
from foo import *


def some_function(...):
    current_time = time.time()
    ...
```

This works fine, your tests pass, you commit the change, and off it goes up the deployment pipeline. Time passes, until one day errors start flooding in. The traceback tells you that your code is causing `AttributeError` exceptions when trying to call `time.time()`. But the unit tests are all green—not only are the tests for your code passing, so are the tests for `foo`.

What's happening in this ripped-from-reality scenario is that someone has added a `time` to `foo.py` that isn't the standard library module of the same name. Maybe they defined `time` as a module-level global variable, or made a function with the same name, or perhaps they, too, didn't like typing a module name in numerous function calls and so imported it like this:

```
from time import time

...
```

Because the `import *` happened after the `import time` in your code, the name `time` is replaced by the one from `from foo import *`, supplanted like a pod person from *Invasion of the Body Snatchers*.

The tests didn't catch this error because they were unit tests, and intentionally isolated the code under test (your module—or in reality, mine) from dependencies that are hard to control. The entire `time` module was mocked out in the tests to allow it to be controlled, and because it was mocked out, it presented exactly the expected interface. And of course the tests for `foo` itself pass—they're verifying that the things inside that module are behaving correctly; it's not their responsibility to check up on what callers are doing with this module.

It's an easy mistake to make. Whoever changed `foo.py` didn't look to see if anyone else was importing everything, and you were busy working on other things and either didn't see the change come in or it didn't register with you. This is especially possible if `foo.py` is some third-party library that you might upgrade without much review—this hazard is just a `pip install -U` away!

So, do as PEP-8 suggests, and avoid wildcard imports! Don't do them, don't let your colleagues do them, and if you see them in code that you maintain, correct them.

TIP

If people on your team really like to `import *` and you're having a hard time convincing them that it's a bad idea, just slip this little gem into a module that they import everything from:

```
False, True = True, False
```

Yes, Python will let you reverse the meanings of `True` and `False`. Please use this knowledge with kindness… unless they really deserve it.

TIP

You can also insulate clients from `import *` problems to some degree by defining an `__all__` in your packages—a list of strings that will be the only things imported when someone imports `*` from the module. For example, if we only wanted to let `foo` and `bar` be wildcard-imported, and disallow `baz`, we might write:

```
__all__ = ['foo', 'bar']

def foo():
    ...

def bar():
    ...

def baz():
    ...
```

Overbroadly Silencing Exceptions

If you'd really like to make your future self cry, try throwing a `try`/`except` around some statements and just ignore any exceptions that might be raised:

```
def get_important_object():
    try:
        data = talk_to_database(...)
        return ImportantObject(data)
    except:
        pass

foo = get_important_object()
foo.do_something_important()
```

In this example, `foo` is either an `ImportantObject` instance, in which case it will dutifully `do_something_important` when asked, or it will be None, and blow up with an AttributeError when we try to call that method.

Around my office, we call this the "Diaper Pattern", which is probably my favorite anti-pattern name of all time, because it's vividly apt to anyone who has ever managed the excretory needs of a small human. In our code, we've wrapped a metaphorical diaper around something that might make a mess in the hopes that it will simply catch everything, but as parents of young children know, "blowouts" are inevitable.

It's recently been referred to as "the most diabolical" Python anti-pattern, and for good reason: all the precious context for the actual error is being trapped in the diaper, never to see the light of day or the inside of your issue tracker.

When the "blowout" exception occurs later on, the stack trace points to the location where the secondary error happened, not to the actual failure inside the try block. If we're very lucky, as in the trivial example above, the secondary error happens close to its true source inside the diaper. In the harsh and unforgiving real world, however, the source may be tens, hundreds, or even thousands of lines away, buried under umpteen layers of abstractions in a different module or, worse, in some third-party library, and when it fails, just as diapers fail, it will fail in the small hours of the night when we would rather be sleeping, and someone important will make an awful fuss about it. It will not be fun.

The Zen of Python tells us that "errors should never pass silently… unless explicitly silenced." We're often a lot better off if we just let exceptions bubble up immediately, uncaught, because at least we know what they really are and where they came from. If we can't get our important data, or make our important object, we'll know right away, and why:

```
def get_important_object():
    data = talk_to_database(...)
    return ImportantObject(data)

foo = get_important_object()
foo.do_something_important()
```

If we really do want to catch *some* exceptions, we should catch them explicitly, while allowing unanticipated exceptions to raise freely:

```python
def get_important_object():
    try:
        data = talk_to_database(...)
    except IOError:
        # Handle the exception appropriately;
        # perhaps use default values?
        data = { ... }
    return ImportantObject(data)
```

TIP

Just using a bare `except` isn't a great idea in general, because that will catch *everything*, including exceptions like `SystemExit` and `KeyboardInterrupt` that we're generally not meant to silence. If you must broadly catch exceptions, it's usually better to use `except Exception` instead of just `except`.

The one acceptable use I have found for the "Diaper Pattern" is at high-level system boundaries, such as a RESTful web service that should always return a well-formed JSON response, or an XML-RPC backend that must always return a well-formed blob of XML. In such a case, we *do* want to catch everything that might go wrong, and package it up in a way that clients will be able to cope with it.

But even when the need for the diaper is legitimate, it's not enough to just package up the error and wash our hands of it, because while our clients will be able to complain to us that our service is broken, we won't actually know what the problem is unless we do something to record the exception. The `logging` module that comes with Python makes this so trivial that it's almost pleasant. Assuming you have a `logger` instance handy, simply invoking its `exception` method will log the full stack trace:

```python
def get_important_object():
    ...
    try:
        data = talk_to_database(...)
        return ImportantObject(data)
    except Exception:
        logger.exception("informative message")
        ...
```

Now we'll know what happened, where it happened, and when, and that can make all the difference when cleaning things up in the middle of the night.

Reinventing the Wheel

Maybe it's because the Zen of Python admonishes us that there should be "one and only one obvious way to do it," or maybe it's just because Python makes it so much fun to *make* things, but it's really tempting to reinvent solutions that have come before—in some cases, many, many times. This can lead to some problematic surprises.

In the bad old days when my workplace used Python 1.5, we needed to log things, so someone wrote a `log` module. But `log` was insufficient, so someone else created `xlog`, which was apparently too complicated, so we made `simplelog`. Sensing a lack of awesomeness in `simplelog`, we called upon `superlog` to solve our problems. But `superlog` wasn't performant enough, so we created the minimalist `pylogger`. It, however, was too terse for larger logging payloads, so it seemed like a good idea to write `prettylog` (spoiler alert—it wasn't). Unfortunately, `prettylog` emitted a format that was so difficult to use that we ended up writing a program called `unuglify.py` that we could pipe `prettylog`-formatted output through just so that we could read the darn things. Oh, and it could give you a *completely* different format if you created it with the wrong flag:

Example 5-1. PrettyLog output

```
|-----------------------------------------------------------------|
|V|0622234435|          test.py|            test.py|   26632|   16|
|-----------------------------------------------------------------|
|V|Hello world. This is a PrettyLog log message, and it does some |
|V|funky things when it gets sufficiently long. Good luck parsing |
|V|this stuff! Ops folks go crazy because of this one weird trick!|
|V|None                                                           |
```

Example 5-2. Surprise! Another flavor of PrettyLog output

```
-------------------------------------------------------------------
Level: V
Time: 06/22/2015 23:44:35
Application: test.py
Module: test.py
Line Number: 15
PID: 26632
Hello world. This is a PrettyLog log message, and it does some funk
y things when it gets sufficiently long. Good luck parsing this stu
ff! Ops folks go crazy because of this one weird trick!
```

We were conservative about migrating to Python 2, so while the rest of the world moved on, we backported an early version of the standard library's `logging` module into our codebase, and this slightly-dodgy, poorly maintained port hung around a lot longer than anyone expected it to. Someone who didn't like the interface of `logging` wrote an alternative called `logmanager`, and it also survived for quite some time in roughly equal mindshare with `logging`. Finally, when we deployed Python 2.3, I discovered the standard library's built-in `logging` module and promptly wrote a wrapper around it called `logwrangler` so that it could be easily driven from configuration files. Someone thought that was a nice idea but decided that it should live somewhere else, so it was copied and forked as `agi.log`, where it picked up bugs of its own.

No two of these implementations had exactly identical interfaces, so it was always a surprise to work with code that was trying to do some logging, as any `log`, `logger`, or `objLog` could be any one of nearly a dozen incompatible things. Behaviors, performance characteristics, and even implementation quirks and bugs varied, so it was never clear which one you really wanted to use. One particularly subtle nuance was that some of the `logging` variants used slightly different strategies for determining the calling frame, so they'd be inconsistent about recording the module or line number that a logging call originated from. Others had pre-baked formats that switched up the order of the elements being logged, to the aggravation of the ops folks who had to parse them. It was usually easier to forgo logging entirely to avoid being paralyzed by the choices available.

There are a few key lessons we can take away from this parade of embarrassments:

- Look at the standard library and PyPI to see if someone has already solved your problem. If whatever problem it is isn't your core domain, your home-grown implementation is probably going to be worse.
- If you decide to replace one solution in favor of another, see it through; update the codebase to your new standard so that you don't have a dozen ways to do essentially the same thing.
- Establish and enforce standards so that surprises—both during development and at runtime—are minimized.

Mutable Keyword Argument Defaults

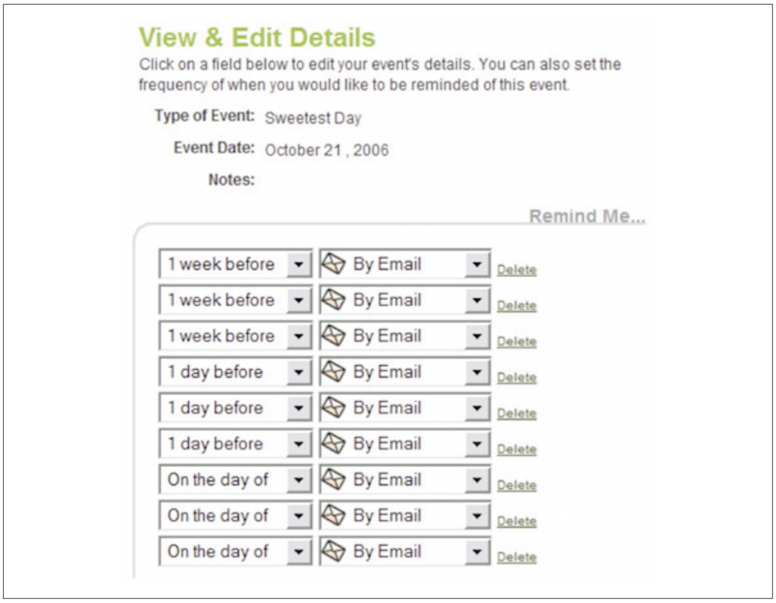
Let's say you want one of the arguments to a function to be optional, and to have a default value when you don't explicitly pass one. Python makes it easy—just use a keyword argument, like so:

```
def foo(bar, baz=42):
    ...
```

But if the default value happens to be something mutable—a list, dictionary, class, object, that sort of thing—you may have some surprises in store for you if you end up altering it in any way. Consider this well-intentioned function from a real-world web application that wants to set some reminders associated with an event:

```
def set_reminders(self, event, reminders=[]):
    reminders.extend(self.get_default_reminders_from_prefs())
    for reminder in reminders:
        ...
```

This function tries to be so helpful: it's trying to self-document that it takes a list of reminder objects, it's trying to let that argument be optional, and it's trying to ensure the user's preferences are respected. But what it's really doing is setting the stage for mysterious bug reports from QA that are accompanied by screenshots like this:



View & Edit Details

Click on a field below to edit your event's details. You can also set the frequency of when you would like to be reminded of this event.

Type of Event: Sweetest Day

Event Date: October 21 , 2006

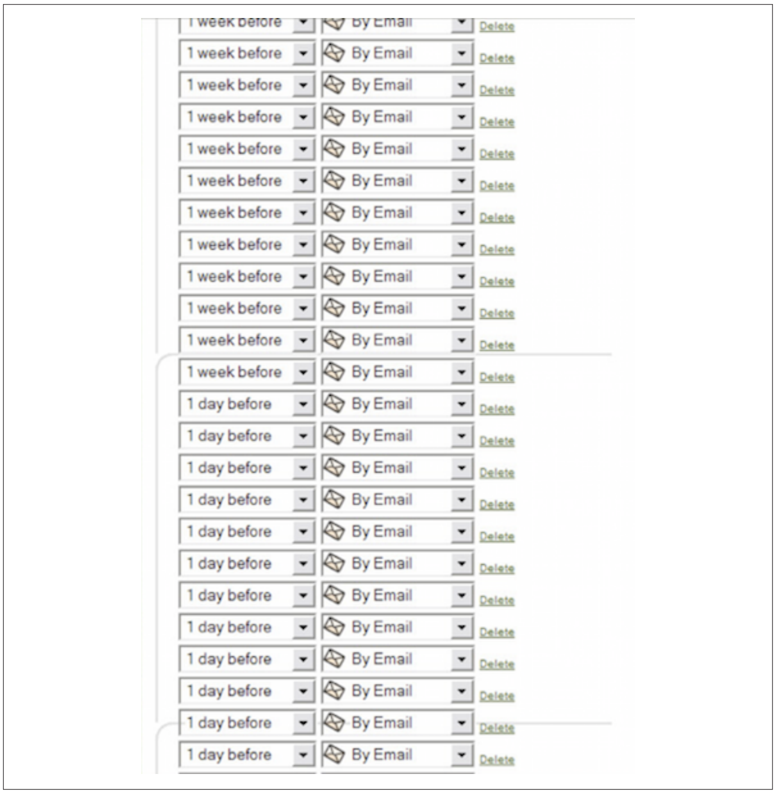
Notes:

Remind Me...

1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
On the day of	By Email	Delete
On the day of	By Email	Delete
On the day of	By Email	Delete

Figure 5-1. Way too many reminders!

And, after a few more hours of QA testing, this:



1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 week before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete
1 day before	By Email	Delete

Figure 5-2. Way, way too many reminders!

What the heck happened? Why is this such a train wreck? Once the right logging was introduced, the problem became clear.

It turns out that Python doesn't give us a new list every time the function gets called; it creates that empty list when the function is defined during the import of its module. After that, we get the *same* list every time the function is called without passing `reminders`. This issue never manifested in our development environment because our homegrown app server would fork a fresh Python process for each request. But in our QA and production environments, the app server process was long-lived, so it would look okay for a few minutes immediately after deploying updates with our attempts to smoke out this bug, but would eventually accumulate more and more data as this function handled more traffic.

Stop for a second and really think about that—this excess data came from other requests. Other requests potentially came from other users.

What if it included personal details, like names or addresses? What if it included health information? What if it included payment data, like a credit card number? The consequences could range from embarassing to catastrophic.

Thankfully, this bug never made it to production, and once we located the problem code, the fix was straightforward. When the default value really does need to be mutable, we set it to `None` in the function definition, and then immediately give it a reasonable default inside the function itself:

```python
def set_reminders(self, event, reminders=None):
    reminders = reminders or []
    # or, if there are valid falsey inputs
    # that we'd like to preserve:
    reminders = [] if reminders is None else reminders
    ...
```

This way we're guaranteed to start with a fresh instance each time the function is called, and data won't leak between invokations.

Overeager Code

For longer than I care to admit, I thought it was good for code to be proactive, to make things convenient to set up and get going. This led me to create code that would do too much, too soon, resulting in side effects that hampered reusability and impacted performance.

These mistakes fall into two basic categories: doing too much when a module is imported and doing too much when an object is instantiated.

At Import Time

It can be tempting to set up globally available values at the module level so that everything else can use them right away. Maybe we're establishing a database connection, perhaps performing some expensive calculation, traversing or transforming a large data structure, or fetching data from an external service.

```
""" A module full of useful things! """

db_conn, cursor = connect_to_db(...)

fibs = [fibonacci(x) for x in range(100)]

pairs = requests.get(human_genome_url).json()
for pair in pairs:
    pair = [pair[1], pair[0]]

def simple_function():
    return 1 + 1

...
```

This (admittedly hyperbolic) example illustrates several of these ideas. If we want to call that simple function in another module, even if it doesn't need a database, we're forced to connect to one when this module is imported. We're going to calculate all those Fibonacci numbers, or make a call to a slow web API to get and subsequently mangle giant gobs of data. Any code that wants to use this simple function is stuck waiting for all of this other work to happen. Unit tests that import this module will be disappointingly slow, leading to a temptation to skip or disable them.

What if the calling code doesn't even have a database it can connect to? Or it runs in an environment that can't connect out to the web? That simple little function becomes useless if one of these things fails; the entire module will be impossible to import. Older versions of Python made hunting down these kinds of surprises even more fun, as an exception at import time would get masked in an `ImportError`, so you might not even know what the actual failure was.

A closely related anti-pattern is to put functionality into the `__init__.py` of a package. Imagine a `foo` package that contains an `__init__.py` as well as submodules `bar` and `baz`. Importing from `bar` or `baz` means that Python first imports the `__init__.py`.

This can go wrong in a couple of ways. First, an exception during the import of `__init__.py` will prevent the import of any of the submodules or their contents:

```
""" __init__.py for package 'foo' """

raise Exception("Oh no!")
```

Another possible disaster is a circular import. In this case, nothing from the foo package can be imported because the __init__.py can't import from bar, because it can't import from foo.__init__, which can't import from bar, which can't import from foo.__init__ (and so forth):

```
""" __init__.py for package 'foo' """

from bar import function_from_bar

def my_function():
    return function_from_bar() + 1


""" foo.bar """

from foo import my_function

def function_from_bar():
    ...
```

The takeaways here should be straightforward:

- Don't do expensive things at import.
- Don't couple to resources that might not be available at import.
- Don't put anything into an __init__.py that could jeopardize the import.
- Beware of circular imports.

In short: try not to do that.

At Instantiation Time

Loading up the __init__ or __new__ methods of a class with a lot of extra work is similar to what we saw above at the module level, but with a couple of insidious differences.

First, unless we've made a module-level mess, the import behavior won't be impacted. It may be enticing, daring us to use it even if it has weird dependencies. "After all," says the wicked little voice, "if we're really desperate we can just feed it Mocks or @patch our sorrows away. Come on—it'll be *fun*." If there aren't dependency issues, the class practically double-dog dares us.

Second, if there's any kind of serious performance impact hiding in the `__init__` or `__new__` methods, the system gets to feel it every time an object is instantiated. The danger here is that you'll never notice during development; your brain can't discern miliseconds from microseconds during limited testing. Only when you're working with the class at scale will you be greeted with the surprise of greatly diminished speed. Even when it doesn't *look* like there's much work happening, there can be a *lot* actually taking place.

Let me tell you the story about how I laid the groundwork for a minor instantiation disaster. I was building the backend of that reminder system, and we had decided that it would use simple data transfer objects to shovel data back and forth via XML-RPC. I thought I would be smart and learn from my post-Java getter-and-setter nightmare classes, eschewing their bloated, 40-plus-parameter `__init__` methods in favor of something clean and declarative, like this:

```
class Calendar(DataObject):
    calendar_id = None
    user_id = None
    label = None
    description = None
```

Alas, Python's XML-RPC library only serializes the *instance* attributes of an object, *not* the *class* attributes, meaning that any attribute we hadn't explicitly set on an instance on the backend simply wouldn't exist when it got to the frontend, and vice versa. To avoid having to clutter up the code with `get` and `getattr` calls, we made the parent `DataObject` class do some magic in the `__new__` to copy all of the class attributes into instance attributes as the object was instantiated. To avoid having to create and maintain those overblown `__init__` methods, I made `DataObject` magically sweep up all its keyword arguments and set the corresponding attributes.

This worked well and saved me a ton of typing. But I was uneasy about allowing *all* the keyword arguments to be used to set attributes in the `DataObject` instance, so I created a `StrictDataObject` subclass that would enforce that only expected attributes were set.

Before long I got worried about one day wanting a `DataObject` whose default attributes might have mutable values like lists and dictionaries, defined on the class in that clean, declarative style. Caution was required to ensure that data wouldn't leak between objects in those shared class attributes. Thinking myself very clever indeed, I created the `MutantDataObject`, which carefully made instance copies of mutable class attributes.

Time passed. `MutantDataObject` became popular for its convenience and worked its way into a number of our systems. Everyone was happy until one day when we got a nasty surprise from a new system we were building: the system was so slow that requests were hitting our 30-second fcgi timeout, bringing the website to its knees.

As we poked around, we eventually discovered that we were simply making way too many `MutantDataObject` instances. One or two weren't terrible, but some inefficient logic had us accidentally making and discarding N^2 or N^3 of them. For our typical data sets, this absolutely killed the CPU—the higher the load went, the worse each subsequent request became. We did a little comparative timing analysis on a box that wasn't busy dying, spinning up some minimal objects with only a few class attributes.

`DataObject` was kind of mediocre, and `StrictDataObject` was, predictably, a little bit slower still. But all the magic in `MutantDataObject` blew the timing right through the roof! Don't pay too much attention to the numbers in Figure 5-3, as they weren't captured on current hardware; instead, focus on their relative magnitudes.

Fixing the flawed plumbing that led to instantiating so many objects was off the table due to the time and effort it required, so we resorted to even darker magic to resolve this crisis, creating a new `DataObject` which called upon the eldritch powers of metaclasses to more efficiently locate and handle mutables in the `__new__`. The result was uncomfortably complicated, maybe even Lovecraftian in its horror, but it did deliver signficant performance results (see Figure 5-4).

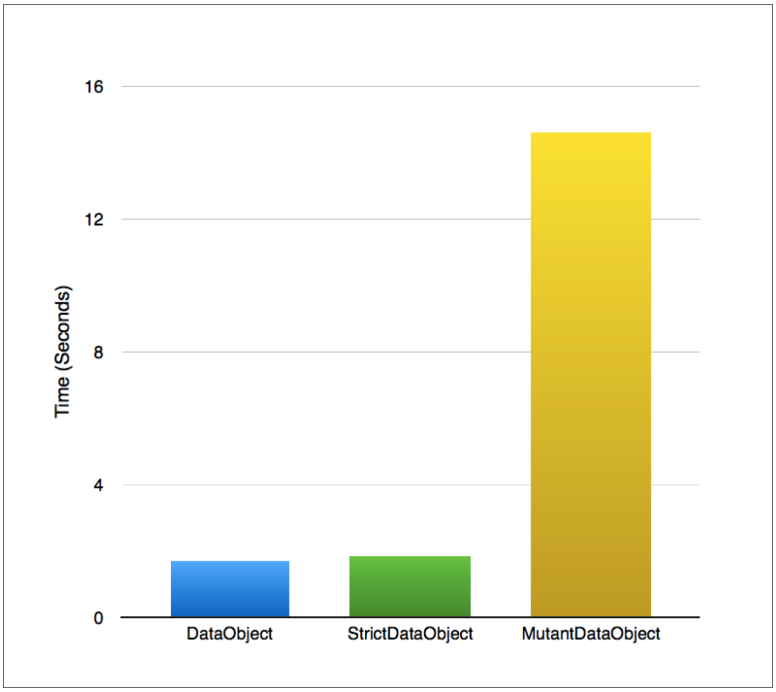


Figure 5-3. Time to instantiate 100,000 objects

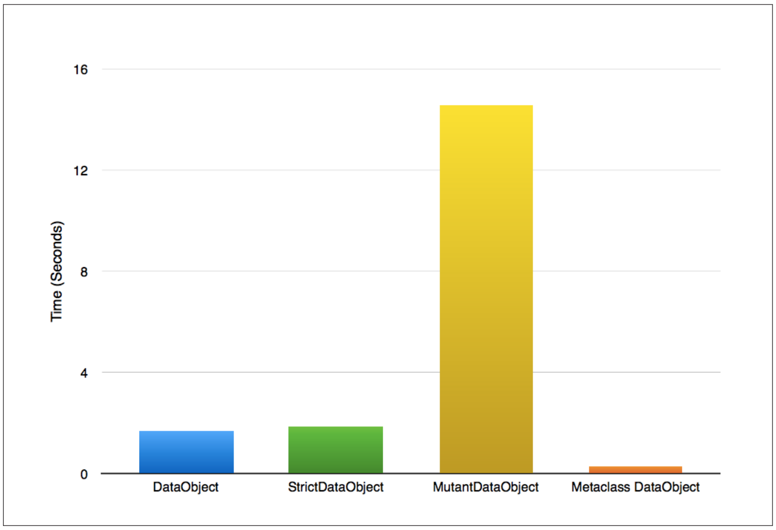


Figure 5-4. Time to instantiate 100,000 objects, revisited

Though we solved the immediate performance problem, we ended up increasing our technical debt by creating both a complicated solution (metaclasses are typically a warning sign) and a maintenance need to phase out the old, naïve implementations in favor of the replacement, plus all of the the attendant QA cost associated with such deeply rooted changes. The victory was pyrrhic at best.

Poisoning Persistent State

Here's another fun mystery. Let's say you've just finished work on an awesome feature. And you've been disciplined about how you executed: you wrote tests along the way, you made sure there was good coverage, you made sure to run them before committing, and all the tests in the parts of the codebase that you touched are passing. You're feeling great…until your CI environment starts barking at you for breaking the build.

So you see if you can reproduce the failure, first rerunning the tests around your changes (nope, still green), and then running the entire test suite, which does show failures inside your tests. Huh? What gives?

What's likely going on is that some other ill-behaved test is sabotaging you. Something, somewhere, is doing some monkey-patching—altering or replacing the contents of a module, class, function, or other object at runtime—and not cleaning up after itself. The test that does this might pass, but causes yours to break as a side effect.

When I first grappled with this scenario, the culprit was a coworker's creation, the aptly named `DuckPuncher` (because Python is "duck typed"):

```
class DuckPuncher(object):
    def __init__(...): ...
    def setup(...): ...
    def teardown(...): ...
    def punch(...): ...
    def hug(...): ...

    def with_setup(self, func):
        def test_func_wrapper(*args, **kwargs):
            self.setup()
            ret = func(*args, **kwargs)
            self.teardown()
            return ret
        test_func_wrapper = wraps(func)(test_func_wrapper)
        return test_func_wrapper
```

Tests that used `DuckPuncher` would inherit from it, define a `setup` and `teardown` that would, respectfully, "punch" (to do the monkey patch) and "hug" (to undo the monkey patch) the metaphorical ducks in question, and `with_setup` would be applied as a decorator around a method that would execute the test, the idea being that the actual test would automatically have the setup and teardown happen around it. Unfortunately, if something fails during the call to the wrapped method, the `teardown` never happens, the punched ducks are never hugged, and now the trap is set. Any other tests that make use of whatever duck was punched will get a nasty surprise when they expect to use the real version of whatever functionality was patched out.

Maybe you're lucky and this hurts immediately—if a built-in like `open` was punched, the test runner (Nose, in my case), will die immediately because it can't read the stack trace generated by the test failure. If you're unlucky, as in our mystery scenario above, it may be 30 or 40 directories away in some vastly unrelated code, and only methodically trying different combinations of tests will locate the real problem. It's even more fun when the tests that are breaking are for code that hasn't changed in six months or more.

A better, smarter `DuckPuncher` would use `finally` to make sure that no matter what happens during the wrapped function, the `teardown` is executed:

```
class DuckPuncher(object):
    def __init__(...): ...
    def setup(...): ...
    def teardown(...): ...
    def punch(...): ...
    def hug(...): ...

    def with_setup(self, func):
        def test_func_wrapper(*args, **kwargs):
            self.setup()
            try:
                ret = func(*args, **kwargs)
            finally:
                self.teardown()
            return ret

        test_func_wrapper = wraps(func)(test_func_wrapper)
        return test_func_wrapper
```

However, this still relies on someone remembering to hug every punched duck; if the `teardown` is omitted, is incomplete, or has its own exception, the test run has still been poisoned. We will instead be much happier if we get comfortable with Mock and its `patch` decorator and context manager. These mechanisms allow us to seamlessly monkey-patch just the items we need to mock out during the test, confident that it will restore them as we exit the context of the test:

```
from mock import patch

from my_code import MyThing

class TestMyThing(...):

    @patch('__builtin__.open'):
    def test_writes_files(self, mock_open):
        ...

    @patch('my_code.something_it_imported'):
    def test_uses_something_imported(self, mock_thing):
        ...
```

As an added bonus, using Mock means that we don't have to reinvent any wheels.

This kind of problem isn't limited to testing. Consider this example from the reminder system discussed earlier:

```
DCT_BRAND_REMINDERS = {
    SITE_X: {
       HOLIDAYS: [Reminder(...), ...],
       OTHER: [Reminder(...), ...],
       CUSTOM: [Reminder(...), ...],
    }, ...
}

...

class BrandWrangler(object):
    ...
    def get_default_reminders(self, brand):
        return DCT_BRAND_REMINDERS.get(brand, {})
```

In this module, I laid out a dictionary of default reminders for different flavors of event, configured for each site that the system supports. The `get_default_reminders` method would then fetch the right set of defaults for a given brand. It went horribly wrong, of course, when the code that needed these defaults would then stamp the `Reminder` instances with things like the user ID or the ID of whatever event the reminder was associated with, causing more data to leak between users across different requests.

When you're being clever about making configuration in code like this, it's a bad idea to give callers the original objects. They're better off with copies (in this case using `deepcopy` so that every object in that subdictionary is fresh and new):

```
import copy

...

class BrandWrangler(object):
    ...
    def get_default_reminders(self, brand):
        return copy.deepcopy(
                DCT_BRAND_REMINDERS.get(brand, {}))
```

Any time you're messing with the contents of a module, or of a class definition, or of anything else that persists outside the scope of a function call, you have an opportunity to shoot yourself in the foot. Proceed with caution when you find yourself writing code like this, and make good use of logging to verify that your assumptions hold.

Assuming Logging Is Unnecessary

Being the intelligent, attractive, and astute reader that you are, you may have noticed a bit of a theme emerging around the notion of logging. This is not coincidence; logging is one of our greatest allies in the struggle against surprises. It is also something that, for various reasons, I have been absolutely terrible at.

I'm a big fan of excuses like:

- "This code is too simple to need logging."
- "The service I'm integrating with will always work."
- "I'll add logging later."

Maybe some of these sound familiar to you?

These excuses are rooted in well-meaning, pure-hearted optimism, a sincere belief that everything will be okay, that we're good enough and smart enough. However, I cannot even begin to count the number of times that this starry-eyed laziness has been my undoing.

The code's too simple? Baloney—code will pile up, something will eventually go wrong, and it'll be hard to diagnose. Integrating with a third-party service? Your code might be golden, but can you prove it? And what product owner is going to prioritize the work to add logging over whatever hot new feature they're really excited to launch? The only way you're adding logging later is when you have to because something's gone horribly wrong and you have no idea what or where.

Having good logging is like having an army of spies arranged strategically throughout your code, witnesses who can confirm or deny your understandings and assumptions. It's not very exciting code; it doesn't make you feel like a ninja rockstar genius. But it will save your butt, and your future self will thank you for being so considerate and proactive.

Okay, so you're determined to learn from my failures and be awesome at logging. What should you be thinking about? What differentiates logging from logging *well*?

Log at Boundaries

Logging fits naturally at boundaries. That can be when entering or leaving a method, when branching (`if`/`elif`/`else`) or looping (`for`, `while`), when there might be errors (`try`/`except`/`finally`), or before and after calling some external service. The type of boundary will guide your choice of log level; for example, `debug` is best in branching and looping situations, where `info` makes more sense when entering or leaving larger blocks. (More on this shortly.)

Log Actions, Motives, and Results

Logging helps you understand the story of your code at runtime. Don't just log *what* you're doing, but *why*, and *what happened*. This can include actions you're about to take, decisions made and the information used to make them, errors and exceptions, and things like the URL of a service you're calling, the data you're sending to it, and what it returned.

Log Mindfully

It's not a good idea to just log indiscriminately; a little bit of mindfulness is important.

Unless you have a fancy aggregation tool, like Splunk or Loggly, a single application (or website) should share a single log file. This makes it easier to see everything that the application is doing, through every layer of abstraction. Dependency injection can be profoundly helpful here, so that even shared code can be provided with the right log.

Choose an appropriate level when logging messages. Take a moment to really think about whether a message is for debugging, general information, a caution, or an error. This will help you to filter the logged data when you're sifting through it. Here are some illustrative suggestions, assuming the standard library's `logging` interface:

```
# Debug - for fine details, apply liberally
log.debug("Initializing frobulator with %s",
        frobulation_values)

# Info - for broader strokes
log.info("Initiating frobulation!")

# Warn - when we're not in trouble yet but
#        should proceed with caution
log.warn("Using a deprecated frobulator; you're "
        "on your own...")

# Error - when something bad has happened
log.error("Unable to frobulate the prognostication "
        "matrix (Klingons?)")

# Exception - when an exception has been raised
#             and we'd like to record the stack trace
log.exception("Unable to frobulate the prognostication "
        "matrix!")

# Critical - when a fatal error has happened and
#            we cannot proceed
log.critical("Goodbye, cruel world!")
```

But we also have to be careful that we don't log things we shouldn't. In particular, be mindful of unsanitized user input, of users' personally identifiable information, and especially health and payment data, as HIPAA and PCI incidents are one hundred percent No Fun At All. You might consider wrapping any sensitive data in another object (with an opaque `__str__` and `__repr__`) so that if it is accidentally logged, the value is not inappropriately emitted.

Assuming Tests Are Unnecessary

The only thing that's bitten me as badly as forgoing decent logging has been skimping on writing tests, or skipping them altogether. This is another place that, as with logging, I have a tendency to assume that the code is too simple to be wrong, or that I can add tests later when it's more convenient. But whenever I say one of these things to myself, it's like a dog whistle that summons all manner of bugs directly and immediately to whatever code isn't being tested.

A recent reminder of the importance of testing came as I was integrating with a third-party service for delivering SMS messages. I had designed and written all the mechanisms necessary for fulfilling the industry and governmental regulations for managing user opt-ins, rate limiting, and record keeping, and somewhere along the way had come to the conclusion that I didn't need to test the integration with the messaging service, because it would be too complicated and wouldn't provide much value since I surely had gotten everything right the first time. This bad assumption turned into weeks of painful manual integration testing as each mistake I uncovered had to be fixed, reviewed, merged, and redeployed into the testing environment. Eventually I reached my breaking point, took a day to write the tests I should have written in the first place, and was amazed by how quickly my life turned from despair to joy.

Python gives us great power and freedom, but as the wisest scholars tell us, we must temper these with *responsibility*:

> With great power comes great responsibility.
>
> —Benjamin "Uncle Ben" Parker

> Right now, we've got freedom *and* responsibility. It's a very groovy time.
>
> —Austin Powers

As long as our code is syntactically reasonable, Python will cheerfully do its best to execute it, even if that means we've forgotten to return a value, gotten our types mismatched, mixed up a sign in some tricky math, used the wrong variable or misspelled a variable name, or committed any number of other common programmer errors. When we have unit tests, we learn about our errors up front, as we make them, rather than during integration—or worse, production—where it’s much more expensive to resolve them. As an added bonus, when your code can be easily tested, it is more likely to be better structured and thus cleaner and more maintainable.

So go make friends with `unittest`, Pytest, or Nose, and explore what the Mock library can do to help you isolate components from one another. Get comfortable with testing, practice it until it becomes like a reflex. Be sure to test the failure conditions as well as the “happy path,” so that you know that when things fail, they fail in the correct way. And most importantly, factor testing into all your estimates, but never as a separate line item that can be easily sacrificed by a product owner or project manager to achieve short-term gains. Any extra productivity squeezed out in this way during the initial development is really borrowed from the future with heavy interest.

Testing now will help prevent weird surprises later.

CHAPTER 6

Further Resources

Education never ends, Watson. It is a series of lessons with the greatest for the last.
—Sherlock Holmes

Now that you've seen many flavors of mistakes, here are some ideas for further exploration, so that you can make more interesting mistakes in the future.

Philosophy

PEP-8
: The definitive resource for the Python community's standards of style. Not everyone likes it, but I enjoy how it enables a common language and smoother integration into teams of Python programmers.

The Zen of Python
: The philosophy of what makes Python *pythonic*, distilled into a series of epigrams. Start up a Python shell and type `import this`. Print out the results, post them above your screen, and program yourself to dream about them.

The Naming of Ducks
: Brandon Rhodes' PyCon talk about naming things well.

The Little Book of Python Anti-Patterns
: A recent compilation of Python anti-patterns and worst practices.

Getters/Setters/Fuxors
: One of the inspirational posts that helped me better understand Python and properties.

Freedom Languages
: An inspirational post about "freedom languages" like Python and "safety languages" like Java, and the mindsets they enable.

Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin (Prentice-Hall, 2008)
: "Uncle Bob" Martin's classic text on code smells and how to progressively refactor and improve your code for readability and maintainability. I disagree with the bits about comments and inline documentation, but everything else is spot-on.

Head First Design Patterns by Eric Freeman and Elizabeth Robson, with Kathy Sierra and Bert Bates (O'Reilly, 2004)
: Yes, the examples are all in Java, but the way it organically derives principles of good object-oriented design fundamentally changed how I thought. There's a lot here for an eager Pythonista.

Tools

Python Editors
: Links to some editors that may make your life easier as a Python developer.

Nose
: Nose is a unit testing framework that helps make it easy to write and run unit tests.

Pytest
: Pytest is a unit testing framework much like Nose but with some extra features that make it pretty neat.

Mock
: Lightweight mock objects and patching functionality make it easier to isolate and test your code. I give thanks for this daily.

Pylint
: *The* linter for Python; helps you detect bad style, various coding errors, and opportunities for refactoring. Consider rigging this

up to your source control with a pre-commit hook, or running it on your code with a continuous integration tool like Jenkins or Travis CI.

Virtualenv
: Virtual environments allow you to work on or deploy multiple projects in isolation from one another; essential for your sanity.

Virtualenvwrapper
: Provides some nice convenience features that make it easier to spin up, use, and work with virtual environments. Not essential, but nice.

Conda
: A package and environment management system for those times when `pip` and `virtualenv` aren't enough.

IPython and Jupyter Notebook
: IPython is the command-line shell and the kernel of the Jupyter Notebook, the browser-based Python environment that enables exploration, experimentation, and knowledge sharing in new and exciting ways. The Notebook has profoundly changed the way I work.

About the Author

Mike Pirnat is an Advisory Engineer at social expression leader American Greetings, where he's wrangled Python since 2000. He's been deeply involved in PCI and security efforts, developer education, and all manner of web development. He is also the cochair of AG's annual Hack Day event.

He has spoken at several PyCons, PyOhios, and CodeMashes and was a cohost and the producer of *From Python Import Podcast* before its long slumber began. (Like the Norwegian Blue, it's only resting.)

He tweets as *@mpirnat* and occasionally blogs at *mike.pirnat.com*.