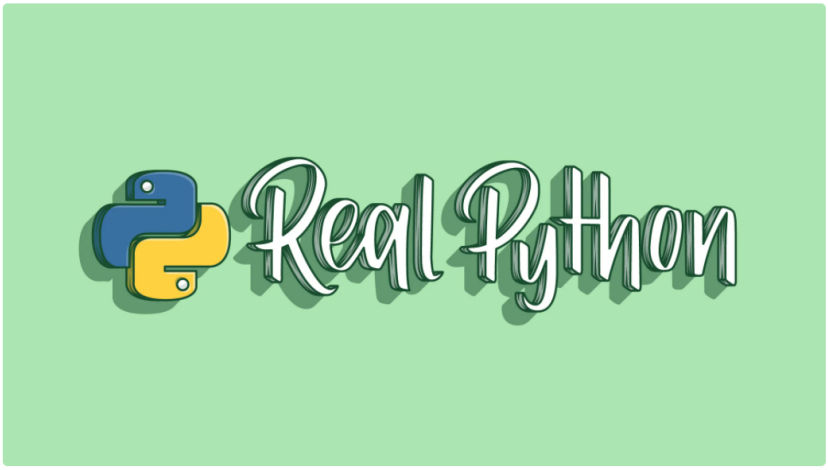


The Most Diabolical Python Antipattern

by Real Python Jan 27, 2015 50 Comments best-practices intermediate python

Mark as Completed

Table of Contents

The following is a guest post by Aaron Maxwell, author of Powerful Python.

There are plenty of ways to write bad code. But in Python, one in particular reigns as king.

We were exhausted, yet jubilant. After two other engineers had tried for *three days each* to fix a mysterious Unicode bug before giving up in vain, I finally isolated the cause after a mere day's work. And ten minutes later, we had a candidate fix.

The tragedy is that we could have skipped the seven days and gone straight to the ten minutes. But I'm getting ahead of myself…

Here's the punchline. The following bit of code is one of the most self-destructive things a Python developer can write:

Python

```
try:
    do_something()
except:
    pass
```

There are variants that amount to the same thing—saying `except Exception:` or `except Exception as e:`, for example. They all do the same massive disservice: silently and invisibly hiding error conditions that can otherwise be quickly detected and dispatched.

Table of Contents
- Why Do We Do This To Ourselves?
- The Solutions
- What You Can Do Now
- Why Log The Full Stack Trace?

Mark as Completed
Tweet Share Email

Why do I claim this is the most diabolical anti-pattern in the Python world today?

- People do this because they expect a specific error to happen there. However, catching `Exception` hides *all* errors… even those which are completely unexpected.
- When the bug is finally discovered - too often, because it has shown up in production - you may have little or no idea where in the code base it's gone wrong. It can take you a truly disheartening amount of time to figure out the error is even happening in that try block.
- Once you realize the error is happening there, you are greatly hampered in your troubleshooting by a lack of critical information. What was the error/exception class? What call or data structure was involved? What line of code, and in what file, did the error originate from?
- You are throwing away the stack trace - a *literally priceless* body of information that can make the difference between troubleshooting a bug in days, or *minutes*. Yes, minutes. More on this later.
- Worst of all, this can easily end up harming the morale, happiness, and even self-esteem of the engineers working on the code base. When the error rears its head, the troubleshooter can sink hours into understanding the root cause alone. They think they are a bad programmer because it takes so long to figure out. They are not; errors which arise by silently catching Exception are intrinsically hard to identity, troubleshoot and fix.

In my nearly ten years of experience writing applications in Python, both individually and as part of a team, this pattern has stood out as the single greatest drain on developer productivity and application reliability… especially over the long term. If you think you have a candidate for something worse, I'd love to hear it.

Why Do We Do This To Ourselves?

Of course, no one *deliberately* writes code designed to stress out your fellow developers and sabotage the reliability of the application. We do this because the code in the try block is expected to sometimes fail in normal operation, in some specific way. Optimistically trying then catching an exception is an excellent and fully Pythonic way to approach this situation.

Insidiously, to catch Exception then silently continue doesn't seem like all that horrible an idea in the moment. But as soon as you save the file, you've set up your code to create the worst kinds of bugs:

- Bugs that can escape detection during development, and be pushed out to the live production system.
- Bugs that can live in production code for minutes, hours, days or weeks before you realize the bug has been happening the whole time.
- Bugs that are hard to troubleshoot.
- Bugs that are hard to fix even once you know where the suppressed exception is being raised.

Note that I am NOT saying never to catch Exception. There *are* good reasons to catch Exception, and then continue - just not *silently*. A good example is a mission-critical process you simply don't want to ever go down. There, a smart pattern is to inject try clauses that catch Exception, log the full stack trace at severity `logging.ERROR` or greater, and continue.

The Solutions

So if we don't want to catch Exception, what do we do instead? There are two choices.

In most cases, the best choice is to catch a more specific exception. Something like this:

Python

```
try:
    do_something()
# Catch some very specific exception - KeyError, ValueError, etc.
except ValueError:
    pass
```

This is the first thing you should try. It takes a little bit of understanding of the invoked code, so you know what types of errors it might raise. This is easier to do well when you're first writing the code, as opposed to cleaning up someone else's.

If some code path simply must broadly catch all exceptions - for example, the top-level loop for some long-running persistent process - then each caught exception *must* write the **full stack trace** to a log or file, along with a timestamp. If you are using Python's `logging` module, this is very easy - each logger object has a method called exception, taking a message string. If you call it in the except block, the caught exception will automatically be fully logged, including the trace.

Python

```
import logging

def get_number():
    return int('foo')
try:
    x = get_number()
except Exception as ex:
    logging.exception('Caught an error')
```

The log will contain the error message, followed by a formatted stack trace spread across several lines:

Python Traceback

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 8, in <module>
    x = get_number()
  File "example-logging-exception.py", line 5, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```

Very easy.

What if your application does logging some other way - not using the `logging` module? Assuming you don't want to refactor your application to do so, you can just fetch and format the traceback associated with the exception. This is easiest in Python 3:

Python

```
# The Python 3 version. It's a little less work.
import traceback

def log_traceback(ex):
    tb_lines = traceback.format_exception(ex.__class__, ex, ex.__traceback__)
    tb_text = ''.join(tb_lines)
    # I'll let you implement the ExceptionLogger class,
    # and the timestamping.
    exception_logger.log(tb_text)

try:
    x = get_number()
except Exception as ex:
    log_traceback(ex)
```

In Python 2, you have to do slightly more work, because exception objects don't have their traceback attached to them. You get this by calling `sys.exc_info()` in the except block:

Python

```
import sys
import traceback

def log_traceback(ex, ex_traceback):
```

```
    tb_lines = traceback.format_exception(ex.__class__, ex, ex_traceback)
    tb_text = ''.join(tb_lines)
    exception_logger.log(tb_text)

try:
    x = get_number()
except Exception as ex:
    # Here, I don't really care about the first two values.
    # I just want the traceback.
    _, _, ex_traceback = sys.exc_info()
    log_traceback(ex, ex_traceback)
```

As it turns out, you can define a single traceback-logging function that will work for both Python 2 and 3:

Python

```
import traceback

def log_traceback(ex, ex_traceback=None):
    if ex_traceback is None:
        ex_traceback = ex.__traceback__
    tb_lines = [ line.rstrip('\n') for line in
                 traceback.format_exception(ex.__class__, ex, ex_traceback)]
    exception_logger.log(tb_lines)
```

What You Can Do Now

“Okay Aaron, you’ve convinced me. I weep and grieve for all the times I have done this in the past. How can I atone?” I’m so glad you asked. Here are a few practices you can start today.

Explicitly Prohibit It In Your Coding Guidelines

If your team does code reviews, you may have a coding guidelines document. If not, it’s easy to start - this can be as simple as creating a new wiki page, and your first entry can be this one. Just add the following two guidelines:

- If some code path simply must broadly catch all exceptions - for example, the top-level loop for some long-running persistent process - then each such caught exception *must* write the **full stack trace** to a log or file, along with a timestamp. Not just the exception type and message, but the full stack trace.
- For all other except clauses - which really should be the vast majority - the caught exception type must be as specific as possible. Something like KeyError, or ConnectionTimeout, etc.

Create Tickets For Existing Overbroad Except Clauses

The above will help prevent new problems from making it into your code base. What about the existing overbroad catches? Simple: make a ticket or issue in your bug tracking system to fix it. This is an easy action step that greatly increases the chances it will be solved and not forgotten about. Seriously, you can do it *right now*.

I recommend you proceed by making a single ticket for each repository or application, to audit through the code to find every place Exception is caught. (You may be able to find them all by just grepping over the code base for “except:” and “except Exception”.) For each occurrence, either convert it to catch a very specific exception type; or if it’s not immediately clear what that should be, instead modify the except block to log the full stack trace.

Optionally, the auditing developer can create additional tickets for any specific try/except block. This is a good thing to do if you have a feeling the exception class can be made more specific, but don’t know that part of the code well enough to be confident. In that case, you

put in code to log the full stack trace; create a separate ticket to investigate further; and assign it to someone who might be more clear. If you find yourself spending more than five minutes thinking about a specific try/except block, I recommend you do this and move on to the next.

Educate Your Fellow Team Members

Do you hold regular engineering meetings? Weekly, fortnightly, or monthly? Ask for five minutes at the next one to explain this antipattern, the cost it has to your productivity as a team, and the simple solutions.

Even better, go to your tech lead or engineering manager beforehand and tell them about it. That will be a much easier sell, since they are at least as concerned about the productivity of the team as you are. Send them the link to this essay. Heck, if you have to, I will help - get them on the phone with me, and I will convince them.

You can reach even wider in your community. Do you go to a local Python meetup? Do they have lightning talks, or can you otherwise negotiate five to fifteen minutes of speaker time at the next meeting? Serve your fellow engineers by evangelizing this noble cause.

Why Log The Full Stack Trace?

Several times above, I have harped on logging the full stack trace, and not just the exception object's message. If this seems like more work, that's because it can be: the trace has newlines that can mess with your logging system's formatting, you may have to muck with the traceback module, and so on. Isn't logging just the message itself enough?

No, it's not. A well-crafted exception message only tells you where the except clause is - what file and what line of code. It commonly doesn't even narrow it down that much, but let's assume the best case here. Logging just the message is better than not logging anything, but unfortunately it does not tell you anything about where the error originates. In general, it can be in a completely different file or module, and often it's not very easy to guess.

Beyond this, real applications developed by teams tend to have multiple code paths that can call the exception-raising block. Maybe the error occurs only when method bar of class Foo is called, but never when function bar() is called. Logging just the message won't help you discern between these two.

The best war story I have is from working on a medium-sized engineering team, around fifty strong. I was relatively new, and was handed a unicode bug that was regularly waking up whoever was on-call for over four months. The exception was caught, and the message logged, but no other info was recorded. Two more senior engineers had worked on it for days each, and then given up, saying that they couldn't figure it out.

These were formidable and scary smart engineers, too. Finally, out of desperation, they tried passing it to me. Using their extensive notes, I immediately set on reproducing the issue well enough to get a stack trace. And after six hours, I finally got it. Once I had that bleeping stack trace, can you guess how long it took me to have a fix?

Ten minutes. That's right. Once we had a stack trace, the fix was obvious. A **literal week** of engineer time could have been saved if we had been logging stack traces from the beginning. Remember above, when I say a stack trace can make the difference between solving a bug in days and solving it in minutes? I wasn't kidding.

(Interestingly, something good came out of it. It's experiences like this that led me to start writing more about Python, and how we as engineers can be more effective with the language.)

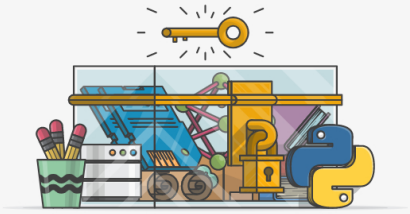
ever. Unsubscribe any time. Curated by the Real Python team.

```
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate python