Raising Hell, Catching Errors

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

One of my favorite Python topics to talk about is error handling—specifically, how to use and not use exceptions. Error handling is hard and tricky. Error handling can mean the difference between an application that can be debugged and one that can't. Error handling can blow your business up in the middle of the night if you aren't careful. So, yes, how you handle errors is important. In this article, I'll dig into some of the details of exceptions, some surefire techniques for shooting yourself in the foot, and some ways to avoid it.

Exception Handling Basics

To signal errors, Python almost always uses exceptions. For example:

```
>>> int('42')
42
>>> int('fortytwo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'fortytwo'
>>>
```

If you want to catch an exception, use the `try-except` statement to enclose a block of code where a failure might occur. For example:

```
def spam(s):
   try:
      x = int(s)
      print('Value is', x)
   except ValueError as e:
      print('Failed: Reason %s' % e)
```

```
>>> spam('42')
Value is 42
>>> spam('fortytwo')
Failed: Reason invalid literal for int() with base 10: 'fortytwo'
>>>
```

Exceptions always have an associated value that is an instance of the exception type. The "`as e`" clause on the `except` captures this instance and puts it into a local variable `e`. If you print out the value, you'll usually just get the error message.

If you want to catch different kinds of errors, you can have multiple `except` blocks. For example:

```
try:
   ...
```

```
except ValueError as e:
    ...
except TypeError as e:
    ...
```

Or, if you want to group the exceptions together so that they're handled by the same except block, use a tuple:

```
try:
    ...
except (ValueError, TypeError) as e:
    ...
```

Certain functions in Python don't raise exceptions but rely on return codes instead. Such functions are rare, but one such example is the `find()` method of strings, which returns -1 if no match can be found:

```
>>> s = 'Hello World'
>>> s.find('Hell')
0
>>> s.find('Cruel')    # No match
-1
>>>
```

The end-of-file condition on files and sockets is also signaled by returning an empty string instead of raising an exception. For example:

```
>>> f = open('somefile.txt', 'r')
>>> data = f.read()      # Read all of the data
>>> f.read()            # Read at EOF
''
>>>
```

Again, such examples of using return codes are rare. Most of the time errors are indicated through exceptions, and catching an exception is a straightforward process using `try-except`.

How to Indicate an Error

In your own code, don't be shy about raising exceptions. If there is some kind of problem, use the `raise` statement to announce it:

```
def spam(x):
    if x < 0:
        raise ValueError('x must be >= 0')
    ...
```

A common mistake made by newcomers is to indicate errors in some other way. For example, with a `print` statement and maybe a special return code:

```
def spam(x):
    if x < 0:
        print('x must be >= 0')
        return None
    ...
```

There are all sorts of problems with such an approach. First, the output of the `print()` is easily lost or overlooked. Moreover, it's pretty likely that other code won't be expecting the `None` return code and won't check for it. Thus, the error might just disappear into the void. This can make for an interesting debugging session later. No, it is almost always better to loudly announce errors with the `raise` statement. That is the Python way—embrace it.

Another immediate problem with raising an exception concerns the exception type that you're supposed to use. Python predefines about two dozen built-in exception types that are always available (i.e., `NameError`, `ArithmeticError`, `IOError`, etc.). Most of these errors are most applicable to Python itself, but a few specific exceptions might be useful in application code. A `ValueError` is commonly used to indicate a bad value as shown. Raise a `TypeError` if you want to signal an error related to bad types (e.g., a list was expected, but the caller passed in a tuple). A generic `RuntimeError` is available to indicate other kinds of problems.

In larger applications, it may make more sense to define your own hierarchy of exceptions instead of relying on the built-ins. This is easily done using classes and inheritance. For example, you start by making a new top-level exception like this:

```
class MyError(Exception):
    pass
```

You can then make more specific kinds of errors that inherit from `MyError`:

```
class MyAuthenticationError(MyError):
    pass
class MyIntegrityError(MyError):
    pass
class MyTimeoutError(MyError):
    pass
class MyBadValueError(MyError):
    pass
```

Use the `raise` statement as before to indicate the exact error that you want:

```
def spam(x):
    if x < 0:
        raise MyBadValueError('x must be >= 0')
    ...
```

One advantage of defining your own hierarchy of exceptions is that you can more easily isolate errors originating from your application as opposed to those from Python itself or any third-party libraries you might have installed. You simply catch the top-level exception like this:

```
try:
    ...
except MyError as e:
    # Catches any exception that subclasses MyError
    ...
```

Isolating your exceptions can be a useful tactic for debugging. For example, if the code dies from a built-in exception, it might indicate a bug in your code, whereas code that dies due to one of your custom exceptions might indicate a bug in someone else's code (e.g., whoever is calling your code). By being more precise about exceptions, you can more easily assign blame when things go wrong—you want that.

What Exceptions Should You Catch?

Given that exceptions are the preferred way of indicating errors, what exceptions are you supposed to catch in your code anyway? It might seem counterintuitive, but I almost never write code to catch exceptions—instead, I simply let them propagate out, possibly causing the program to crash. As an example, consider this code fragment:

```
def parse_file(filename):
    f = open(filename)
    ...
```

Now, suppose that the user passes a bad filename and the `open()` function fails with an `IOError` exception. You could write the code to account for that possibility by wrapping the `open()` with a `try-except` like this:

```
def parse_file(filename):
    try:
        f = open(filename)
    except IOError as e:
        # Handle the error in some way  ???
        ...
    ...
```

However, if you do this, it suddenly raises all sorts of questions. For example, what are you supposed to do in the `except` block? Do you print a message? Do you raise an exception? If you raise an exception, how is it any different from `open()` raising an `IOError`? Last but not least, even if the code catches the error, is there any way that the function can proceed afterwards? If there is no file, there is nothing to parse. How would it work?

As a rule of thumb, you should probably never catch exceptions unless your code can sensibly recover and take action in some way. Just to illustrate, a much more likely scenario would be a parsing function that needed to account for bad values:

```
def parse_file(filename):
    f = open(filename)
    for line in f:
        fields = line.split()
        try:
            x = float(fields[0])
        except ValueError:
            x = float('nan')
        ...
```

Here, catching a possible exception and using it to take corrective action makes sense. These are the kinds of errors you should be concerned with—not errors for which there is no hope of sane recovery. Put another way, if something is going to fail spectacularly and there's no hope, it's often better to step back and let it fail. Don't let your code get mixed up in the middle of the mess.

Beware the Diaper Pattern

Now wait just a minute—surely I can't be advocating a coding style where you never catch errors. Python code might be running some kind of important service where it can't just crash and disappear with a traceback. It's important to catch the nuance of the previous section. Basically, you shouldn't be writing code that attempts to catch exceptions for which no recovery is possible *at that point*. The possibility of a sane recovery really depends on context. For example, a parsing function clearly can't continue if it can't read data. However, if that function was invoked from within a larger framework executing a request on behalf of a client in a distributed system, there might be code that broadly catches failures and reports them back to the client.

A common technique for broad exception handling is to enclose code in a `try-except` block like this:

```
try:
    ...
    statements
    ...
except Exception as e:       # Catch any error
    # Handle the failure
    ...
```

`Exception` is the root of all error-related exceptions in Python (note: certain exceptions such as `SystemExit` derive from `BaseException` and won't be caught here). Thus, this code will catch any programming error that might occur.

This is the so-called "diaper pattern" in action—code that catches anything. It's also one of the most dangerous exception-handling approaches to be using. Don't be the programmer that writes code like this:

```
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no')
    return
```

Or worse yet:

```
try:
    ...
    statements
    ...
except Exception as e:
    # TODO: Whoops, it failed.
    pass
```

Such code is the fastest way to create an undebuggable program. Any failure whatsoever, including common programming errors such as a misspelled variable name, will simply result in a vague error message. You'll never be able to figure out what's wrong.

If you're going to catch all errors, it is imperative that you report the actual reason for the failure. At a minimum, you might do this:

```
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no. Reason %s' % e)
    return
```

A much better alternative is to make the full traceback available somewhere. If it's not going to be reported directly to the end-user for some reason, it can be written to a log file using the `logging` module and the inclusion of the `exc_info=True` option to logging functions. For example:

```
import logging
log = logging.getLogger('mylog')

...
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no.')
    log.error('FAILURE: %s', e, exc_info=True)
    return
```

Alternatively, you can produce the traceback message yourself and obtain it as a string using the `traceback` module. This can be useful if you want to do something with the traceback such as redirect it elsewhere (e.g., to an email message, a database, etc.). For example:

```
import traceback
try:
    ...
    statements
    ...
except Exception as e:
    print('Computer says no.')
    tb = traceback.format_exc()    # Create the traceback message
    # Do something with tb
    ...
    return
```

It should be noted that the traceback module has additional functions for pulling apart stack traces and formatting them. Consult the online documentation [1] for more information.

If your intent is to merely log the error while allowing it to propagate, you can use a bare `raise` statement to re-raise the exception. For example:

```
try:
    ...
except Exception as e:
    log.error('FAILURE: %s', e, exc_info=True)
    raise    # Reraise the exception
```

You, Yes You Did It!

As much as you might try to sanely handle errors in your code, dealing with errors in large systems is still tricky. One particularly nasty problem arises if you capture exceptions and then raise a different kind of exception to encapsulate the "failure" in some broad way. For example, consider the following code:

```
class OperationalError(Exception):
    pass

def run_function(func, *args, **kwargs):
    try:
        return func(*args, **kwargs)
    except Exception as e:
        raise OperationalError('Function failed. Reason %s' % e)
```

Now, watch what happens in Python 2:

```
>>> def add(x, y):
...     return x + y
...
>>> run_function(add, 2, 3)
5
```

```
>>> run_function(add, 2, '3')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in run_function
__main__.OperationalError: Function failed. Reason
unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Here, you get an error message and it has some details about the failure, but the information is woefully incomplete. In particular, the traceback contains no useful information about what actually happened in the `add()` function itself.

Chained exceptions [2] is one area where Python 3 shines. If you try this same code on Python 3, you get two tracebacks:

```
>>> run_function(add, 2, '3')
Traceback (most recent call last):
  File "<stdin>", line 3, in run_function
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in run_function
__main__.OperationalError: Function failed. Reason
unsupported operand type(s) for +: 'int' and 'str'
>>>
```

This is exception chaining in action. You can further refine the exact error message by adding a `from` modifier to the `raise` statement like this:

```
class OperationalError(Exception):
    pass

def run_function(func, *args, **kwargs):
    try:
        return func(*args, **kwargs)
    except Exception as e:
        raise OperationalError('Function failed') from e
```

Now, the error message changes to the following:

```
>>> run_function(add, 2, '3')
Traceback (most recent call last):
  File "<stdin>", line 3, in run_function
  File "<stdin>", line 2, in add
TypeError: unsupported operand type(s) for +: 'int' and 'str'

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in run_function
__main__.OperationalError: Function failed
>>>
```

In this case, you're seeing a chain of exceptions and information about causation. That's pretty nice.

Final Thoughts (and Advice)

Proper handling of errors is an important aspect of any application. However, you want to make sure you do it in a way that allows you to maintain your sanity. The following rules of thumb provide a summary of some of the ideas in this article:

- Prefer the use of exceptions to indicate errors. It is the most common Python style and will be less error prone than alternatives such as returning special codes from functions.
- Don't write code that catches exceptions from which no sensible recovery is possible. It's better to simply let the exception propagate to some other code that knows how to deal with the error.
- Be extremely careful when writing code that catches all errors. Make sure you always report diagnostic information somewhere where it can be found by developers. Otherwise, you'll quickly end up with undebuggable Python code.

As an aside, a recent article in *;login:* about catastrophic failures in distributed systems [3] reported that nearly 35% of these problems were caused by trivial mistakes in exception handling. Although not specific to Python, that article is definitely worth a read.

References

[1] https://docs.python.org/2/library/traceback.html (Traceback Module).

[2] https://www.python.org/dev/peps/pep-3134/ (Exception Chaining).

[3] D. Yuan et al., "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," *;login:*, vol. 40, no. 1, February 2015 (USENIX).

Donate Today: The USENIX Annual Fund

Many USENIX supporters have joined us in recognizing the importance of open access over the years. We are thrilled to see many more folks speaking out about this issue every day. If you also believe that research should remain open and available to all, you can help by making a donation to the USENIX Annual Fund at www.usenix.org/annual-fund.

With a tax-deductible donation to the USENIX Annual Fund, you can show that you value our Open Access Policy and all our programs that champion diversity and innovation.

The USENIX Annual Fund was created to supplement our annual budget so that our commitment to open access and our other good works programs can continue into the next generation. In addition to supporting open access, your donation to the Annual Fund will help support:

- USENIX Grant Program for Students and Underrepresented Groups
- Special Conference Pricing and Reduced Membership Dues for Students
- Women in Advanced Computing (WiAC) Summit and Networking Events
- Updating and Improving Our Digital Library

With your help, USENIX can continue to offer these programs—and expand our offerings—in support of the many communities within advanced computing that we are so happy to serve. Join us!

We extend our gratitude to everyone that has donated thus far, and to our USENIX and LISA SIG members; annual membership dues helped to allay a portion of the costs to establish our Open Access initiative.

www.usenix.org/annual-fund