Write Cleaner Python: Use Exceptions

Many programmers have had it drilled into their head that exceptions, in any language, should only be used in truly exceptional cases. They're wrong. The Python community's approach to exceptions leads to cleaner code that's easier to read. And that's without the monstrous hit to performance commonly associated with exceptions in other languages.

EDIT: Updated with more useful exception idioms

Using exceptions to write *cleaner* code?

When I talk about "using exceptions", I'm specifically *not* referring to creating some crazy exception hierarchy for your package and raising exceptions at every possible opportunity. That will most certainly lead to unmaintainable and difficult to understand code. This notion has been widely discussed and is well summarized on Joel Spolsky's blog.

Note: Python avoids much of the tension of the "error codes vs exceptions" argument. Between the ability to return multiple values from a function and the ability to return values of different types (e.g. `None` or something similar in the error case) the argument is moot. But this is besides the point.
The style of exception usage I'm advocating is quite different. In short: **take advantage of Python built-ins and standard library modules that already throw exceptions.** Exceptions are built in to Python at the lowest levels. In fact, I *guarantee* your code is already using exceptions, even if not explicitly.

Intermezzo: How the `for` statement works

Any time you use `for` to iterate over an `iterable` (basically, all `sequence` types and anything that defines `__iter__()` or `__getitem__()`), it needs to know when to stop iterating. Take a look at the code below:

```
words = ['exceptions', 'are', 'useful']
for word in words:
    print(word)
```

How does `for` know when it's reached the last element in `words` and should stop trying to get more items? The answer may surprise you: *the list raises a `StopIteration` exception*.

In fact, all `iterables` follow this pattern. When a `for` statement is first evaluated, it calls `iter()` on the object being iterated over. This creates an `iterator` for the object, capable of returning the contents of the object in sequence. For the call to `iter()` to succeed, the object must either support the iteration protocol (by defining `__iter__()`) or the sequence protocol (by defining `__getitem__()`).

As it happens, both the `__iter__()` and `__getitem__()` functions are required to *raise an exception* when the items to iterate over are exhausted. `__iter__()` raises the `StopIteration` exception, as discussed earlier,

and `__getitem__()` raises the `IndexError` exception. This is how `for` knows when to stop.

In summary: if you use `for` anywhere in your code, you're using exceptions.

LBYL vs. EAFP

It's all well and good that exceptions are widely used in core Python constructs, but *why* is a different question. After all, `for` could certainly have been written to not rely on exceptions to mark the end of a sequence. Indeed, exceptions could have been avoided altogether.

But they exist due to the philosophical approach to error checking adopted in Python. Code that doesn't use exceptions is always checking if it's OK to do something. In practice, it must ask a number of different questions before it is convinced it's OK to do something. If it doesn't ask *all* of the right questions, bad things happen. Consider the following code:

```
def print_object(some_object):
    # Check if the object is printable...
    if isinstance(some_object, str):
        print(some_object)
    elif isinstance(some_object, dict):
        print(some_object)
    elif isinstance(some_object, list):
        print(some_object)
    # 97 elifs later...
    else:
        print("unprintable object")
```

This trivial function is responsible for calling `print()` on an object. If it can't be `print()`-ed, it prints an error message.

Trying to anticipate all error conditions in advance is destined for failure (and is also really ugly). Duck typing is a central idea in Python, but this function will incorrectly print an error for types than *can* be printed but aren't explicitly checked.

The function can be rewritten like so:

```
def print_object(some_object):
    # Check if the object is printable...
    try:
        printable = str(some_object)
        print(printable)
    except TypeError:
        print("unprintable object")
```

If the object can be coerced to a string, do so and print it. If that attempt raises an exception, print our error string. Same idea, much easier to follow (the lines in the `try` block could obviously be combined but weren't to make the example more clear). Also, note that we're explicitly checking for `TypeError`, which is what would be raised if the coercion failed. Never use a "bare" `except:` clause or you'll end up suppressing real errors you didn't intend to catch.

But wait, there's more!

The function above is admittedly contrived (though certainly based on a common anti-pattern). There are a number of other useful ways to use exceptions. Let's take a look at the use of an `else` clause when handling exceptions.

In the rewritten version of `print_object` below, the code in the `else` block is executed only if the code in the `try` block **didn't** throw an exception. It's conceptually similar to using `else` with a `for` loop (which is itself a useful, if not widely known, idiom). It also fixes a bug in the previous version: we caught a `TypeError` assuming that only the call to `str()` would generate it. But what if it was actually (somehow) generated from the call to `print()` and has nothing to do with our string coercion?

```
def print_object(some_object):
    # Check if the object is printable...
    try:
        printable = str(some_object)
    except TypeError:
        print("unprintable object")
    else:
        print(printable)
```

Now, the `print()` line is only called if no exception was raised. If `print()` raises an exception, this will bubble up the call stack as normal. The `else` clause is often overlooked in exception handling but incredibly useful in certain situations. Another use of `else` is when code in the `try` block requires some cleanup (and doesn't have a usable context manager), as in the below example:

```
def display_username(user_id):
    try:
        db_connection = get_db_connection()
    except DatabaseEatenByGrueError:
        print('Sorry! Database was eaten by a grue.')
    else:
        print(db_connection.get_username(user_id))
        db_connection.cleanup()
```

How not to confuse your users

A useful pattern when dealing with exceptions is the bare `raise`. Normally, `raise` is paired with an exception to

be raised. However, if it's used in exception handling code, `raise` has a slightly different (but immensely useful) meaning.

```
def calculate_value(self, foo, bar, baz):
    try:
        result = self._do_calculation(foo, bar, baz)
    except:
        self.user_screwups += 1
        raise
    return result
```

Here, we have a member function doing some calculation. We want to keep some statistics on how often the function is misused and throws an exception, but we have no intention of actually handling the exception. Ideally, we want to an exception raised in `_do_calculation` to be flow back to the user code as normal. If we simply raised a new exception from our `except` clause, the traceback point to our `except` clause and mask the real issue (not to mention confusing the user). `raise` on its own, however, lets the exception propagate normally *with its original traceback*. In this way, we record the information we want and the user is able to see what actually caused the exception.

A tale of two styles

We've now seen two distinct approaches to error handling (lots of `if` statements vs. catching exceptions). These approaches are respectively known as *Look Before You Leap (LBYL)* and *Easier to Ask for Forgiveness than Permission*. In the LBYL camp, you always check to see if something can be done before doing it. In EAFP, you just do the thing. If it turns out that wasn't possible, *shrug* "my bad", and deal with it.

Idiomatic Python is written in the EAFP style (where reasonable). We can do so because exceptions are cheap in Python.

Slow is relative

The fact that the schism over exception usage exists is understandable. In a number of other languages (especially compiled ones), exceptions are comparatively expensive. In this context, avoiding exceptions in performance sensitive code is reasonable.

But this argument doesn't hold weight for Python. There is *some* overhead, of course, to using exceptions in Python. *Comparatively*, though, it's negligible in almost all cases. And I'm playing it safe by including "almost" in the previous sentence.

Want proof? Regardless, here's some proof. To get an accurate sense of the overhead of using exceptions, we need to measure two (and a half) things:

1. The overhead of simply adding a `try` block but never throwing an exception
2. The overhead of using an exception vs. comparable code without exceptions

1. When the exception case is quite likely
2. When the exception case is unlikely

The first is easy to measure. We'll time two code blocks using the `timeit` module. The first will simply increment a counter. The second will do the same but wrapped in a `try`/`except` block.

Here's the script to calculate the timings:

```
SETUP = 'counter = 0'

LOOP_IF = """
counter += 1
"""

LOOP_EXCEPT = """
try:
    counter += 1
except:
    pass
"""



if __name__ == '__main__':
    import timeit
    if_time = timeit.Timer(LOOP_IF, setup=SETUP)
    except_time = timeit.Timer(LOOP_EXCEPT, setup=SETUP)
    print('using if statement: {}'.format(min(if_time.repeat(number=10 ** 7))))
    print('using exception: {}'.format(min(except_time.repeat(number=10 ** 7))))
```

Note that `Timer.repeat(repeat=3, number=1000000)` returns the time taken to execute the code block `number` times, repeated `repeat` times. The Python documentation suggests that the time should be at least 0.2 to be accurate, hence the change to `number`.
The code prints the best run of executing each code block (`LOOP_IF` and `LOOP_EXCEPT`) 10,000,000 times.

Clearly, all we're measuring here is the setup cost of using an exception. Here are the results:

```
>>> python exception_short
using if statement: 0.574051856995
using exception: 0.821137189865
```

So the presence of an exception increases run time by .3 seconds divided by 10,000,000. In other words: **if using a simple exception drastically impacts your performance, you're doing it wrong...**

So an exception that does nothing is cheap. Great. What about one that's actually useful? To test this, we'll load

the words file found at `/usr/share/dict/words` on most flavors of Linux. Then we'll conditionally increment a counter based on the presence of a random word. Here is the new timing script:

```
import timeit

SETUP = """
import random
with open('/usr/share/dict/words', 'r') as fp:
    words = [word.strip() for word in fp.readlines()]
percentage = int(len(words) *.1)
my_dict = dict([(w, w) for w in random.sample(words, percentage)])
counter = 0
"""

LOOP_IF = """
word = random.choice(words)
if word in my_dict:
    counter += len(my_dict[word])
"""

LOOP_EXCEPT = """
word = random.choice(words)
try:
    counter += len(my_dict[word])
except KeyError:
    pass
"""



if __name__ == '__main__':
    if_time = timeit.Timer(LOOP_IF, setup=SETUP)
    except_time = timeit.Timer(LOOP_EXCEPT, setup=SETUP)
    number = 1000000
    min_if_time = min(if_time.repeat(number=number))
    min_except_time = min(except_time.repeat(number=number))

    print """using if statement:
    minimum: {}
    per_lookup: {}
    """.format(min_if_time, min_if_time / number)

    print """using exception:
    minimum: {}
    per_lookup: {}
```

```
    """.format(min_except_time, min_except_time / number)
```

The only thing of note is the `percentage` variable, which essentially dictates how likely our randomly chosen `word` is to be in `my_dict`.

So with a 90% chance of an exception being thrown in the code above, here are the numbers:

```
using if statement:
    minimum: 1.35720682144
    per_lookup: 1.35720682144e-06

using exception:
    minimum: 3.25777006149
    per_lookup: 3.25777006149e-06
```

Wow! 3.2 seconds vs 1.3 seconds! Exceptions are teh sux0rz!

If you run them 1,000,000 times in a tight loop with a 90% chance of throwing an exception, then exceptions are a bit slower, yes. Does any code you've *ever* written do that? No? Good, let's see a more realistic scenario.

Changing the chance of an exception to 20% gives the following result:

```
using if statement:
    minimum: 1.49791312218
    per_lookup: 1.49791312218e-06

using exception:
    minimum: 1.92286801338
    per_lookup: 1.92286801338e-06
```

At this point the numbers are close enough to not care. A difference of 0.5 * 10^-6 seconds shouldn't matter to anyone. If it does, I have a spare copy of the K&R C book you can have; go nuts.

What did we learn?

Exceptions in Python are not "slow".

To sum up...

Exceptions are baked-in to Python at the language level, can lead to cleaner code, and impose almost zero performance impact. If you were hesitant about using exceptions in the style described in this post, don't be. If you've avoided exceptions like the plague, it's time to give them another look.

Posted on Feb 06, 2013 by Jeff Knupp