Python Exceptions

Aaron Maxwell

aaron@powerfulpython.com

Contents

Chapter 1

Exceptions and Errors

Errors happen. That's why every practical programming language provides a rich framework for dealing with them.

Python's error model is based on *exceptions*. Some of you reading this are familiar with exceptions, and some are not; some of you have used exceptions in other languages, and not yet with Python. This chapter is for all of you.

If you are familiar with how exceptions work in Java, C++ or C#, you'll find Python uses similar concepts, even if the syntax is completely different. And beyond those similarities lie uniquely Pythonic patterns.

We'll start with the basics some of you know. Even if you've used Python exceptions before, I recommend reading all of this chapter. Odds are you will learn useful things, even in sections which appear to discuss what you've seen before.

1.1 The Basic Idea

An *exception* is a way to interrupt the normal flow of code. When an exception occurs, the block of Python code will stop executing - literally in the middle of the line - and immediately jump to *another* block of code, designed to handle the situation.

Often an exception means an error of some sort, but it doesn't have to be. It can be used to signal anticipated events, which are best handled in an interrupt-driven way. Let's illustrate the common, simple cases first, before exploring more sophisticated patterns.

You've already encountered exceptions, even if you didn't realize it. Here's a little program using a `dict`:

```
# favdessert.py
def describe_favorite(category):
    "Describe my favorite food in a category."
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    return "My favorite {} is {}.".format(
        category, favorites[category])

message = describe_favorite("dessert")
print(message)
```

When run, this program exits with an error:

```
Traceback (most recent call last):
  File "favdessert.py", line 12, in <module>
    message = describe_favorite("dessert")
  File "favdessert.py", line 10, in describe_favorite
    category, favorites[category])
KeyError: 'dessert'
```

When you look up a missing dictionary key like this, we say Python *raises* a *KeyError*. (In other languages, the terminology is "throw an exception". Same idea; Python uses the word "raise" instead of "throw".) That `KeyError` is an *exception*. In fact, most errors you see in Python are exceptions. This includes `IndexError` for lists, `TypeError` for incompatible types, `ValueError` for bad values, and so on. When an error occurs, Python responds by raising an exception.

An exception needs to be handled. If not, your program will crash. You handle it with *try-except* blocks. They look like this:

```
# Replace the last few lines with the following:
try:
    message = describe_favorite("dessert")
    print(message)
except KeyError:
    print("I have no favorite dessert. I love them all!")
```

Notice the structure. You have the keyword `try`, followed by an indented block of code, immediately followed by `except KeyError`, which has its own block of code. We say the `except` block *catches* the `KeyError` exception.

Run the program with these new lines, and you get the following output:

```
I have no favorite dessert. I love them all!
```

Importantly, the new program exits successfully; its exit code to the operating system indicates "success" rather than "failure".

Here's how `try` and `except` work:

- Python starts executing lines of code in the `try:` block.
- If Python gets to the end of the `try` block and no exceptions are raised, Python skips over the `except` block completely. None of its lines are executed, and Python proceeds to the next line after (if there is one).
- If an exception is raised anywhere in the `try` block, the program immediately stops - literally in the middle of the line; no further lines in the `try` block will be executed. Python then checks whether the exception type (`KeyError`, in this case) matches an `except` clause. If so, it jumps to the matching block's first line.
- If the exception does *not* match the `except` block, the exception ignores it, acting like the block isn't even there. If no higher-level code has an `except` block to catch it, the program will crash.

Let's wrap these lines of code in a function:

```
def print_description(category):
    try:
        message = describe_favorite(category)
        print(message)
    except KeyError:
        print("I have no favorite {}. I love them all!".format(
            category))
```

Notice how `print_description` behaves differently, depending on what you feed it:

```
>>> print_description("dessert")
I have no favorite dessert. I love them all!
>>> print_description("appetizer")
My favorite appetizer is calamari.
>>> print_description("beverage")
My favorite beverage is coffee.
>>> print_description("soup")
I have no favorite soup. I love them all!
```

Exceptions aren't just for damage control. You will sometimes use them as a flow-control tool, to deal with ordinary variations you know can occur at runtime. Suppose, for example, your program loads data from a file, in JSON format. You import the `json.load` function in your code:

```
from json import load
```

`json` is part of Python's standard library, so it's always available. Now, imagine there's an open-source library called `speedyjson`,[1] with a `load` function just like what's in the standard library - except twice as fast. And your program works with BIG json files, so you want to preferentially use the `speedyjson` version when available. In Python, importing something that doesn't exist raises an `ImportError`:

```
# If speedyjson isn't installed...
>>> from speedyjson import load
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ImportError: No module named 'speedyjson'
```

[1] Not a real library, so far as I know. But after this book is published, I'm sure one of you will make a library with that name, just to mess with me.

How can you use `speedyjson` if it's there, yet gracefully fall back on `json` when it's not? Use a try-except block:

```
try:
    from speedyjson import load
except ImportError:
    from json import load
```

If `speedyjson` is installed and importable, `load` will refer to its version of the function in your code. Otherwise you get `json.load`.

A single `try` can have multiple `except` blocks. For example, `int()` will raise a `TypeError` if passed a nonsensical type; it raises `ValueError` if the type is acceptable, but its value can't be converted to an integer.

```
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

More realistically, you might log different error events[2] with different levels of severity:

```
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

If an exception is raised, Python will check whether its type matches the first `except` block. If not, it checks the next. The first matching `except` block is executed, and all others are skipped over entirely - so you will never have more than one of the `except` blocks executed for a given `try`. Of course, if none of them match, the exception continues rising until something catches it. (Or the process dies.)

[2] Especially in larger applications, exception handling often integrates with logging. See the logging chapter for details.

There's a good rule of thumb, which I suggest you start building as a habit now: *put as little code as possible in the* `try` *block*. You do this so your `except` block(s) will not catch or mask errors they should not.

Sometimes you will want to have clean-up code that runs *no matter what*, even if an exception is raised. You can do this by adding a `finally` block:

```
try:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

The code in the `finally` block is *always* executed. If an exception is raised in the `try` block, Python will immediately jump to the `finally` block, run its lines, then raise the exception. If an exception is not raised, Python will run all the lines in the `try` block, then run the lines in the `finally` block. It's a way to say, "run these lines no matter what".

You can also have one (or more) except clauses:

```
try:
    line1
    line2
    # etc.
except FirstException:
    line1
    line2
    # etc.
except SecondException:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

What's executed and when depends on whether an exception is raised. If not, the lines in the `try` block run, followed by the lines in the `finally` block; none of the `except` blocks run. If

an exception *is* raised, and it matches one of the `except` blocks, then the `finally` block runs *last*. The order is: the `try` block (up until the exception is raised), then the matching `except` block, and then the finally block.

What if an exception is raised, but there is no matching except block? The `except` blocks are ignored, because none of them match. The lines of code in `try` are executed, up until the exception is raised. Python immediately jumps to the `finally` block; when its lines finish, only then is the exception raised.

It's important to understand this ordering. When you include a `finally` block, and an exception is raised, the code in the `finally` block interjects itself between the code that could run in the `try` block, and the raising of the exception. A `finally` block is like insurance, for code which *must* run, no matter what.

Here's a good example. Imagine writing control code that does batch calculations on a fleet of cloud virtual machines. You issue an API call to rent them, and pay by the hour until you release them. Your code might look something like:

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Now imagine `running_job.wait()` raises a `socket.timeout` exception (which means the network connection has timed out). This causes a stack trace, and the program crashes, or maybe some higher-level code actually catches the exception.

Regardless, now `fleet.terminate()` is never called. Whoops. That could be *really* expensive.

To save your bank balance (or keep your job), rewrite the code using a `finally` block:

```
fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
finally:
    fleet.terminate()
```

This code expresses the idea: "no matter what, terminate the fleet of rented virtual machines." Even if an error in `fleet.submit_job(job)` or `running_job.wait()` makes the program crash, it calls `fleet.terminate()` with its dying breath.

Let's look at dictionaries again. When working directly with a dictionary, you can use the "if key in dictionary" pattern to avoid a `KeyError`, instead of try/except blocks:

```
# Another approach we could have taken with favdessert.py
def describe_favorite_or_default(category):
    'Describe my favorite food in a category.'
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    if category in favorites:
        message = "My favorite {} is {}.".format(
        category, favorites[category])
    else:
        message = "I have no favorite {}. I love them all!".format(
           category)
    return message

message = describe_favorite_or_default("dessert")
print(message)
```

The general pattern is:

```
# Using "if key in dictionary" idiom.
if key in mydict:
    value = mydict[key]
else:
    value = default_value

# Contrast with "try/except KeyError".
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Many developers prefer using the "if key in dictionary" idiom, or using `dict.get()`. But these aren't always the best choice. They are only options if your code has direct access to the dictionary, for one thing. Maybe `describe_favorite()` is part of a library, and you can't change it. Even if it's open-source, you have better things to do than fork a library every time a function interface isn't convenient. Or maybe `describe_favorite()` is code you control, but you just don't *want* to change it, for any number of good reasons. A try-except block catching `KeyError` solves all these problems, because it lets you handle the situation without modifying any code inside `describe_favorite()` itself.

1.2 Exceptions Are Objects

An exception is an object: an instance of an exception class. `KeyError`, `IndexError`, `TypeError` and `ValueError` are all built-in classes, which inherit from a base class called `Exception`. Writing code like `except KeyError:` means "if the exception just raised is of type `KeyError`, run this block of code."

So far, we haven't dealt with those exception objects directly. And often, you don't need to. But sometimes you want more information about what happened, and capturing the exception object can help. Here's the structure:

```
try:
    do_something()
except ExceptionClass as exception_object:
    handle_exception(exception_object)
```

where *ExceptionClass* is some exception class, like `KeyError`, etc. In the except block,

`exception_object` will be an instance of that class. You can choose any name for that variable; no one actually calls it `exception_object`, preferring shorter names like `ex`, `exc`, or `err`. The methods and contents of that object will depend on the kind of exception, but almost all will have an attribute called `args`. That will be a tuple of what was passed to the exception's constructor. The `args` of a `KeyError`, for example, will have one element - the missing key:

```
# Atomic numbers of noble gasses.
nobles = {'He': 2, 'Ne': 10,
  'Ar': 18, 'Kr': 36, 'Xe': 54}
def show_element_info(elements):
   for element in elements:
       print('Atomic number of {} is {}'.format(
             element, nobles[element]))
try:
    show_element_info(['Ne', 'Ar', 'Br'])
except KeyError as err:
    missing_element = err.args[0]
    print('Missing data for element: ' + missing_element)
```

Running this code gives you the following output:

```
Atomic number of Ne is 10
Atomic number of Ar is 18
Missing data for element: Br
```

The interesting bit is in the `except` block. Writing `except KeyError as err` stores the exception object in the `err` variable. That lets us look up the offending key, by peeking in `err.args`. Notice we could not get the offending key any other way, unless we want to modify `show_element_info` (which we may not want to do, or perhaps *can't* do, as described before).

Let's walk through a more sophisticated example. In the `os` module, the `makedirs` function will create a directory:

```
# Creates the directory "riddles", relative
# to the current directory.
import os
os.makedirs("riddles")
```

By default, if the directory already exists, `makedirs` will raise `FileExistsError`:[3] Imagine you are writing a web application, and need to create an upload directory for each new user.

[3] Python 2 does something different, and more complex. We'll cover that in detail later. For now, keep reading.

That directory should not exist; if it does, that's an error and needs to be logged. Our upload-directory-creating function might look like this:

```
# First version....
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError:
        logging.error(
            "Upload dir for new user already exists")
```

It's great we are detecting and logging the error, but the error message isn't informative enough to be helpful. We at least need to know the offending username, but it's even better to know the directory's full path (so you don't have to dig in the code to remind yourself what `UPLOAD_ROOT` was set to).

Fortunately, `FileExistsError` objects have an attribute called `filename`. This is a string, and the path to the already-existing directory. We can use that to improve the log message:

```
# Better version!
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

Only the `except` block is different. That `filename` attribute is perfect for a useful log message.

1.3 Raising Exceptions

`ValueError` is a built-in exception that signals some data is of the correct type, but its format isn't valid. It shows up everywhere:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

Your own code can raise exceptions, just like `int()` does. You should, in fact, so you have better error messages. (And sometimes for other reasons - more on that later.) You do so with the *raise* statement. The most common form is this:

```
raise ExceptionClass(arguments)
```

For `ValueError` specifically, it might look like:

```
def positive_int(value):
    number = int(value)
    if number <= 0:
        raise ValueError("Bad value: " + str(value))
    return number
```

Focus on the `raise` line in `positive_int`. You simply create an instance of `ValueError`, and pass it directly to `raise`. Really, the syntax is `raise exception_object` - though usually you just create the object inline. `ValueError`'s constructor takes one argument, a descriptive string. This shows up in stack traces and log messages, so be sure to make it informative and useful:

```
>>> positive_int("-3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -3
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -7.0
```

Let's show a more complex example. Imagine you have a Money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __repr__(self):
        'Renders the object nicely on the prompt.'
        return "Money({},{})".format(
            self.dollars, self.cents)
    # Plus other methods, which aren't important to us now.
```

Your code needs to create Money objects from string values, like "$140.75". The constructor takes dollars and cents, so you create a function to parse that string and instantiate Money for you:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

This function[4] works like this:

```
>>> money_from_string("$140.75")
Money(140,75)
>>> money_from_string("$12.30")
Money(12,30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in money_from_string
AttributeError: 'NoneType' object has no attribute 'group'
```

This error isn't clear; you must read the source and think about it to understand what went wrong. We have better things to do than decrypt stack traces. You can improve this function's

[4] It's better to make this a class method of Money, rather than a separate function. That is a separate topic, though; see @classmethod in the object-oriented patterns chapter for details.

usability by having it raise a `ValueError`.

```
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    # Adding the next two lines here
    if match is None:
        raise ValueError("Invalid amount: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

The error message is now much more informative:

```
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in money_from_string
ValueError: Invalid amount: 'Big money'
```

1.4 Catching And Re-raising

In an `except` block, you can re-raise the current exception. It's very simple; just write `raise` by itself, with no arguments:

```
try:
    do_something()
except ExceptionClass:
    handle_exception()
    raise
```

Notice you don't need to store the exception object in a variable. It's a shorthand, exactly equivalent to this:

```
try:
    do_something()
except ExceptionClass as err:
    handle_exception()
    raise err
```

This "catch and release" only works in an `except` block. It requires that some higher-level code will catch the exception and deal with it. Yet it enables several useful code patterns. One is when you want to delegate handling the exception to higher-level code, but also want to inject some extra behavior closer to the exception source. For example:

```
try:
    process_user_input(value)
except ValueError:
    logging.info("Invalid user input: %s", value)
    raise
```

If `process_user_input` raises a `ValueError`, the `except` block will execute the logging line. Other than that, the exception propagates as normal.

It's also useful when you need to execute code before deciding whether to re-raise the exception at all. Earlier, we used a try/except block pair to create an upload directory, logging an error if it already exists:

```
# Remember this? Python 3 code, from earlier.
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

This relies on `FileExistsError`, which was introduced in Python 3. How could you do this in Python 2? Even if you no longer write code in Python 2, it's worth studying the different approach required, as it demonstrates a widely useful exception-handling pattern. Let's take a look.

`FileExistsError` subclasses the more general `OSError`. This exception type has been around since the early days of Python, and in Python 2, `makedirs` simply raises `OSError`. But `OSError` can indicate many problems other than the directory already existing: a lack of filesystem permissions, a system call getting interrupted, even a timeout over a network-mounted filesystem. We need a way to discriminate between these possibilities.

`OSError` objects have an `errno` attribute, indicating the precise error. These correspond to the variable `errno` in a C program, with different integer values meaning different error conditions. Most higher-level languages - including Python - reuse the constant names defined in the C API; in particular, the standard constant for "file already exists" is `EEXIST` (which happens to be set to the number 17 in most implementations). These constants are defined in the `errno` module in Python, so we just type `from errno import EEXIST` in our program.

In versions of Python with `FileExistsError`, the general pattern is:

- Optimistically create the directory, and
- if `FileExistsError` is raised, catch it and log the event.

In Python 2, we must do this instead:

- Optimistically create the directory.
- if `OSError` is raised, catch it.
- Inspect the exception's `errno` attribute. If it's equal to `EEXIST`, this means the directory already existed; log that event.
- If `errno` is something else, it means we don't want to catch this exception here; re-raise the error.

The code:

```
# How to accomplish the same in Python 2.
import os
import logging
from errno import EEXIST
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except OSError as err:
        if err.errno != EEXIST:
            raise
        logging.error("Upload dir already exists: %s",
            err.filename)
```

The only difference between the Python 2 and 3 versions is the "except" clause. But there's a lot going on there. First, we're catching `OSError` rather than `FileExistsError`. But we may or may not re-raise the exception, depending on the value of its `errno` attribute. Basically, a value of `EEXIST` means the directory already exists. So we log it and move on. Any other value indicates an error we aren't prepared to handle right here, so re-raise in order to pass it to higher level code.

1.5 The Most Diabolical Python Anti-Pattern

You know about "design patterns": time-tested solutions to common code problems. And you've probably heard of "anti-patterns": solutions people often choose to a code problem, because it *seems* to be a good approach, but actually turn out to be harmful.

In Python, one antipattern is most harmful of all.

I wish I could not even tell you about it. If you don't know it exists, you can't use it in your code. Unfortunately, you might stumble on it somewhere and adopt it, not realizing the danger. So, it's my duty to warn you.

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

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

Python lets you completely omit the argument to `except`. If you do that, it will catch *every exception*. That's pretty harmful right there; remember, the more pin-pointed your `except` clauses are, the more precise your error handling can be, without sweeping unrelated errors under the rug. And typing `except:` will sweep *every* unrelated error under the rug.

But it's much worse than that, because of the `pass` in the `except` clause. What `except: pass` does is silently and invisibly hide error conditions that you'd otherwise quickly detect and fix.

(Instead of `"except:"`, you'll sometimes see variants like `"except Exception:"` or `"except Exception as ex:"`. They amount to the same thing.)

This creates the **worst kind of bug**. Have you ever been troubleshooting a bug, and just couldn't figure out where in the code base it came from, even after hours of searching, getting

more and more frustrated as the minutes and hours roll by? *This is how you create that in Python.*

I first understood this anti-pattern after joining an engineering team, in an explosively-growing Silicon Valley start-up. The company's product was a web service, which needed to be up 24/7. So engineers took turns being "on call" in case of a critical issue. An obscure Unicode bug somehow kept triggering, waking up an engineer - in the middle of the night! - several times a week. But no one could figure out how to reproduce the bug, or even track down exactly how it was happening in the large code base.

After a few months of this nonsense, some of the senior engineers got fed up and devoted themselves to rooting out the problem. One senior engineer did nothing for *three full days* except investigate it, ignoring other responsibilities as they piled up. He made some progress, and took useful notes on what he found, but in the end did not figure it out. He ran out of time, and had to give up.

Then, a second senior engineer took over. Using those notes as a starting point, he also dug into it, ignoring emails and other commitments for *another three full days* to track down the problem. And he failed. He make progress, adding usefully to the notes. But in the end, he had to give up too, when other responsibilities could no longer be ignored.

Finally, after these six days, they passed the torch to me - the new engineer on the team. I wasn't too familiar with the code base, but their notes gave me a lot to go on. So I dove in on Day 7, and completely ignored everything else for six hours straight.

And finally, late in the day, I was able to isolate the problem to a single block of code:

```
try:
    extract_address(location_data)
except:
    pass
```

That was it. The data in `location_data` was corrupted, causing the `extract_address` call to raise a `UnicodeError`. Which the program then *completely silenced*. Not even producing a stack trace; simply moving on, as if nothing had happened.

After nearly *seven full days* of engineer effort, we pinpointed the error to this one block of code. I un-suppressed the exception, and was almost immediately able to reproduce the bug - with a full and very informative stack trace.

Once I did that, can you guess how long it took us to fix the bug?

TEN MINUTES.

That's right. A full WEEK of engineer time was wasted, all because this anti-pattern somehow snuck into our code base. Had it not, then the first time it woke up an engineer, it would have been obvious what the problem was, and how to fix it. The code would have been patched by the end of the day, and we would all have moved on to bigger and better things.

The cruelty of this anti-pattern comes from how it completely hides *all* helpful information. Normally, when a bug causes a problem in your code, you can inspect the stack trace; identify what lines of code are involved; and start solving it. With The Most Diabolical Python Antipattern (TMDPA), none of that information is available. What line of code did the error come from? Which *file* in your Python application, for that matter? In fact, what was the exception type? Was it a `KeyError`? A `UnicodeError`? Or even a `NameError`, coming from a mis-typed variable name? Was it `OSError`, and if so, what was its `errno`? You don't know. You *can't* know.

In fact, TMDPA often **hides the fact that an error even occurs**. This is one of the ways bugs hide from you during development, then sneak into production, where they're free to cause real damage.

We never did figure out why the original developer wrote `except: pass` to begin with. I think that at the time, `location_data` may have sometimes been empty, causing `extract_address` to innocuously raise a `ValueError`. In other words, if `ValueError` was raised, it was appropriate to ignore that and move on. By the time the other two engineers and I were involved, the code base had changed so that was no longer how things worked. But the broad `except` block remained, like a land mine lurking in a lush field.

So why do people do this? Well, no one *wants* to wreak such havoc in their Python code, of course. People do this because they expect errors to occur in the normal course of operation, in some specific way. They are simply catching too broadly, without realizing the full implications.

So what do you do instead? There are two basic choices. In most cases, it's best to modify the `except` clause to catch a more specific exception. For the situation above, this would have been a much better choice:

```
try:
    extract_address(location_data)
except ValueError:
    pass
```

Here, `ValueError` is caught and appropriately ignored. If `UnicodeError` raises, it propagates and (if not caught) the program crashes. That would have been *great* in our situation. The error

log would have a full stack trace clearly telling us what happened, and we'd be able to fix it in ten minutes.

As a variation, you may want to insert some logging:

```
try:
    extract_address(location_data)
except ValueError:
    logging.info(
        "Invalid location for user %s", username)
```

The other reason people write `except: pass` is a bit more valid. Sometimes, a code path simply must broadly catch all exceptions, and continue running regardless. This is common in the top-level loop for a long-running, persistent process. The problem is that `except: pass` hides all information about the problem, including that the problem even exists.

Fortunately, Python provides an easy way to capture that error event, and all the information you need to fix it. The `logging` module has a function called `exception`, which will log your message *along with the full stack trace of the current exception*. So you can write code like this:

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

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

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 5, in <module>
    x = get_number()
  File "example-logging-exception.py", line 3, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```

This stack trace is *priceless*. Especially in more complex applications, it's often not enough to know the file and line number where an error occurs. It's at least as important to know *how* that

function or method was called... what path of executed code led to it being invoked. Otherwise you can never determine what conditions lead to that function or method being called in the first place. The stack trace, in contrast, gives you everything you need to know.

I wish `"except: pass"` was not valid Python syntax. I think much grief would be spared if it was. But it's not my call, and changing it now is probably not practical. Your only defense is to be vigilant. That includes educating your fellow developers. Does your team hold regular engineering meetings? Ask for five minutes at the next one to explain this antipattern, the cost it has to everyone's productivity, and the simple solutions.

Even better: if there are local Python or technical meetups in your area, volunteer to give a short talk - five to fifteen minutes. These meetups almost always need speakers, and you will be helping so many of your fellow developers in the audience.

There is a longer article explaining this situation at https://powerfulpython.com/blog/the-most-diabolical-python-antipattern/ . Simply sharing the URL will educate people too. And feel free to write your own blog post, with your own explanation of the situation, and how to fix it. Serve your fellow engineers by evangelizing this important knowledge.

Index