
Chapter 32. Exception Basics

This part of the book deals with *exceptions*, which are events that can modify the flow of control through a program. In Python, exceptions are triggered automatically on errors, and they can be triggered and intercepted by your code. They are processed by four statements we'll study in this part, the first of which has two variations (listed separately here) and the last of which was an optional extension until Python 2.6 and 3.0:

`try/except`

Catch and recover from exceptions raised by Python, or by you.

`try/finally`

Perform cleanup actions, whether exceptions occur or not.

`raise`

Trigger an exception manually in your code.

`assert`

Conditionally trigger an exception in your code.

`with/as`

Implement context managers in Python 2.6 and 3.0 (optional in 2.5).

This topic was saved until nearly the end of the book because you need to know about classes to code exceptions of your own. With a few exceptions (pun intended), though, you'll find that exception handling is simple in Python because it's integrated into the language itself as another high-level tool.

Why Use Exceptions?

In a nutshell, exceptions let us jump out of arbitrarily large chunks of a program. Consider the hypothetical pizza-making robot we discussed earlier in the book. Suppose we took the idea seriously and actually built such a machine. To make a pizza, our culinary automaton would need to execute a plan, which we would implement as a Python program: it would take an order, prepare the dough, add toppings, bake the pie, and so on.

Now, suppose that something goes very wrong during the "bake the pie" step. Perhaps the oven is broken, or perhaps our robot miscalculates its reach and spontaneously combusts. Clearly, we want to be able to jump to code that handles such states quickly. As we have no hope of finishing the pizza task in such unusual cases, we might as well abandon the entire plan.

That's exactly what exceptions let you do: you can jump to an exception handler in a single step, abandoning all function calls begun since the exception handler was entered. Code in the exception handler can then respond to the raised exception as appropriate (by calling the fire department, for instance!).

One way to think of an exception is as a sort of structured "super go to." An *exception handler* (`try` statement) leaves a marker and executes some code. Somewhere further ahead in the program, an exception is raised that makes Python jump back to that marker, abandoning any active functions that were called after the marker was left. This protocol provides a coherent way to respond to unusual events. Moreover, because Python jumps to the handler statement immediately, your code is simpler—there is usually no need to check status codes after every call to a function that could possibly fail.

Exception Roles

In Python programs, exceptions are typically used for a variety of purposes. Here are some of their most common roles:

Error handling

Python raises exceptions whenever it detects errors in programs at runtime. You can catch and respond to the errors in your code, or ignore the exceptions that are raised. If an error is ignored, Python's default exception-handling behavior kicks in: it stops the program and prints an error message. If you don't want this default behavior, code a `try` statement to catch and recover from the exception—Python will jump to your `try` handler when the error is detected, and your program will resume execution after the `try`.

Event notification

Exceptions can also be used to signal valid conditions without you having to pass result flags around a program or test them explicitly. For instance, a search routine might raise an exception on failure, rather than returning an integer result code (and hoping that the code will never be a valid result).

Special-case handling

Sometimes a condition may occur so rarely that it's hard to justify convoluting your code to handle it. You can often eliminate special-case code by handling unusual cases in exception handlers in higher levels of your program.

Termination actions

As you'll see, the `try/finally` statement allows you to guarantee that required closing-time operations will be performed, regardless of the presence or absence of exceptions in your programs.

Unusual control flows

Finally, because exceptions are a sort of high-level "go to," you can use them as the basis for implementing exotic control flows. For instance, although the language does not explicitly support backtracking, it can be implemented in Python by using exceptions and a bit of support logic to unwind assignments.[73] There is no "go to" statement in Python (thankfully!), but exceptions can sometimes serve similar roles.

We'll see such typical use cases in action later in this part of the book. For now, let's get started with a look at Python's exception-processing tools.

Exceptions: The Short Story

Compared to some other core language topics we've met in this book, exceptions are a fairly lightweight tool in Python. Because they are so simple, let's jump right into some code.

Default Exception Handler

Suppose we write the following function:

```
>>> def fetcher(obj, index):
...     return obj[index]
...
```

There's not much to this function—it simply indexes an object on a passed-in index. In normal operation, it returns the result of a legal index:

```
>>> x = 'spam'
>>> fetcher(x, 3)                       # Like x[3]
'm'
```

However, if we ask this function to index off the end of the string, an exception will be triggered when the function tries to run `obj[index]`. Python detects out-of-bounds indexing for sequences and reports it by *raising* (triggering) the built-in `IndexError` exception:

```
>>> fetcher(x, 4)                       # Default handler - shell interface
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Because our code does not explicitly catch this exception, it filters back up to the top level of the program and invokes the *default exception handler*, which simply prints the standard error message. By this point in the book, you've probably seen your share of standard error messages. They include the exception that was raised, along with a *stack trace*—a list of all the lines and functions that were active when the exception occurred.

The error message text here was printed by Python 3.0; it can vary slightly per release, and even per interactive shell. When coding interactively in the basic shell interface, the filename is just "<stdin>," meaning the standard input stream. When working in the IDLE GUI's interactive shell, the filename is "<pyshell>", and source lines are displayed, too. Either way, file line numbers are not very meaningful when there is no file (we'll see more interesting error messages later in this part of the book):

```
>>> fetcher(x, 4)                       # Default handler - IDLE GUI interface
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    fetcher(x, 4)
  File "<pyshell#3>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

In a more realistic program launched outside the interactive prompt, after printing an error message the default handler at the top also *terminates* the program immediately. That course of action makes sense for simple scripts; errors often should be fatal, and the best you can do when they occur is inspect the standard error

message.

Catching Exceptions

Sometimes, this isn't what you want, though. Server programs, for instance, typically need to remain active even after internal errors. If you don't want the default exception behavior, wrap the call in a `try` statement to catch exceptions yourself:

```
>>> try:
...     fetcher(x, 4)
... except IndexError:                  # Catch and recover
...     print('got exception')
...
got exception
>>>
```

Now, Python jumps to your *handler* (the block under the `except` clause that names the exception raised) automatically when an exception is triggered while the `try` block is running. When working interactively like this, after the `except` clause runs, we wind up back at the Python prompt. In a more realistic program, `try` statements not only catch exceptions, but also *recover* from them:

```
>>> def catcher():
...     try:
...         fetcher(x, 4)
...     except IndexError:
...         print('got exception')
...     print('continuing')
...
>>> catcher()
got exception
continuing
>>>
```

This time, after the exception is caught and handled, the program resumes execution after the entire `try` statement that caught it—which is why we get the "continuing" message here. We don't see the standard error message, and the program continues on its way normally.

Raising Exceptions

So far, we've been letting Python raise exceptions for us by making mistakes (on purpose this time!), but our scripts can raise exceptions too—that is, exceptions can be raised by Python or by your program, and can be caught or not. To trigger an exception manually, simply run a `raise` statement. User-triggered exceptions are caught the same way as those Python raises. The following may not be the most useful Python code ever penned, but it makes the point:

```
>>> try:
...     raise IndexError                # Trigger exception manually
... except IndexError:
...     print('got exception')
...
got exception
```

As usual, if they're not caught, user-triggered exceptions are propagated up to the top-level default exception handler and terminate the program with a standard error message:

```
>>> raise IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
```

As we'll see in the next chapter, the `assert` statement can be used to trigger exceptions, too—it's a conditional `raise`, used mostly for debugging purposes during development:

```
>>> assert False, 'Nobody expects the Spanish Inquisition!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Nobody expects the Spanish Inquisition!
```

User-Defined Exceptions

The `raise` statement introduced in the prior section raises a built-in exception defined in Python's built-in scope. As you'll learn later in this part of the book, you can also define new exceptions of your own that are specific to your programs. User-defined exceptions are coded with *classes*, which inherit from a built-in exception class: usually the class named `Exception`. Class-based exceptions allow scripts to build exception categories, inherit behavior, and have attached state information:

```
>>> class Bad(Exception):                # User-defined exception
...     pass
...
>>> def doomed():
...     raise Bad()                      # Raise an instance
...
>>> try:
...     doomed()
... except Bad:                           # Catch class name
...     print('got Bad')
...
got Bad
>>>
```

Termination Actions

Finally, `try` statements can say "finally"—that is, they may include `finally` blocks. These look like `except` handlers for exceptions, but the `try`/`finally` combination specifies termination actions that always execute "on the way out," regardless of whether an exception occurs in the `try` block:

```
>>> try:
...     fetcher(x, 3)
... finally:                              # Termination actions
...     print('after fetch')
...
'm'
after fetch
>>>
```

Here, if the `try` block finishes without an exception, the `finally` block will run, and the program will resume after the entire `try`. In this case, this statement seems a

bit silly—we might as well have simply typed the `print` right after a call to the function, and skipped the `try` altogether:

```
fetcher(x, 3)
print('after fetch')
```

There is a problem with coding this way, though: if the function call raises an exception, the `print` will never be reached. The `try`/`finally` combination avoids this pitfall—when an exception does occur in a `try` block, `finally` blocks are executed while the program is being unwound:

```
>>> def after():
...     try:
...         fetcher(x, 4)
...     finally:
...         print('after fetch')
...     print('after try?')
...
>>> after()
after fetch
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
  File "<stdin>", line 2, in fetcher
IndexError: string index out of range
>>>
```

Here, we don't get the "after try?" message because control does not resume after the `try`/`finally` block when an exception occurs. Instead, Python jumps back to run the `finally` action, and then propagates the exception up to a prior handler (in this case, to the default handler at the top). If we change the call inside this function so as not to trigger an exception, the `finally` code still runs, but the program continues after the `try`:

```
>>> def after():
...     try:
...         fetcher(x, 3)
...     finally:
...         print('after fetch')
...     print('after try?')
...
>>> after()
after fetch
after try?
>>>
```

In practice, `try`/`except` combinations are useful for catching and recovering from exceptions, and `try`/`finally` combinations come in handy to guarantee that termination actions will fire regardless of any exceptions that may occur in the `try` block's code. For instance, you might use `try`/`except` to catch errors raised by code that you import from a third-party library, and `try`/`finally` to ensure that calls to close files or terminate server connections are always run. We'll see some such practical examples later in this part of the book.

Although they serve conceptually distinct purposes, as of Python 2.5, we can now mix `except` and `finally` clauses in the same `try` statement—the `finally` is run on the way out regardless of whether an exception was raised, and regardless of whether the exception was caught by an `except` clause.

As we'll learn in the next chapter, Python 2.6 and 3.0 provide an alternative to `try`/`finally` when using some types of objects. The `with`/`as` statement runs an object's

context management logic to guarantee that termination actions occur:

```
>>> with open('lumberjack.txt', 'w') as file:          # Always close file on exit
...     file.write('The larch!\n')
```

Although this option requires fewer lines of code, it's only applicable when processing certain object types, so `try/finally` is a more general termination structure. On the other hand, `with/as` may also run startup actions and supports user-defined context management code.

WHY YOU WILL CARE: ERROR CHECKS

One way to see how exceptions are useful is to compare coding styles in Python and languages without exceptions. For instance, if you want to write robust programs in the C language, you generally have to test return values or status codes after every operation that could possibly go astray, and propagate the results of the tests as your programs run:

```
doStuff()
{                                   # C program
    if (doFirstThing() == ERROR)    # Detect errors everywhere
        return ERROR;               # even if not handled here
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

In fact, realistic C programs often have as much code devoted to error detection as to doing actual work. But in Python, you don't have to be so methodical (and neurotic!). You can instead wrap arbitrarily vast pieces of a program in exception handlers and simply write the parts that do the actual work, assuming all is well:

```
def doStuff():          # Python code
    doFirstThing()      # We don't care about exceptions here,
    doNextThing()       # so we don't need to detect them
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()       # This is where we care about results,
    except:             # so it's the only place we must check
        badEnding()
    else:
        goodEnding()
```

Because control jumps immediately to a handler when an exception occurs, there's no need to instrument all your code to guard for errors. Moreover, because Python detects errors automatically, your code usually doesn't need to check for errors in the first place. The upshot is that exceptions let you largely ignore the unusual cases and avoid error-checking code.

Chapter Summary

And that is the majority of the exception story; exceptions really are a simple tool.

To summarize, Python exceptions are a high-level control flow device. They may be raised by Python, or by your own programs. In both cases, they may be ignored (to trigger the default error message), or caught by `try` statements (to be processed by your code). The `try` statement comes in two logical formats that, as of Python 2.5, can be combined—one that handles exceptions, and one that executes finalization code regardless of whether exceptions occur or not. Python's `raise` and `assert` statements trigger exceptions on demand (both built-ins and new exceptions we define with classes); the `with`/`as` statement is an alternative way to ensure that termination actions are carried out for objects that support it.

In the rest of this part of the book, we'll fill in some of the details about the statements involved, examine the other sorts of clauses that can appear under a `try`, and discuss class-based exception objects. The next chapter begins our tour by taking a closer look at the statements we introduced here. Before you turn the page, though, here are a few quiz questions to review.

Test Your Knowledge: Quiz

1. Name three things that exception processing is good for.
2. What happens to an exception if you don't do anything special to handle it?
3. How can your script recover from an exception?
4. Name two ways to trigger exceptions in your script.
5. Name two ways to specify actions to be run at termination time, whether an exception occurs or not.