CHAPTER 8

Exceptions

When writing computer programs, it is usually possible to discern between a normal course of events and something that's exceptional (out of the ordinary). Such exceptional events might be errors (such as trying to divide a number by zero) or simply something you might not expect to happen very often. To handle such exceptional events, you might use conditionals everywhere the events might occur (for example, have your program check whether the denominator is zero for every division). However, this would not only be inefficient and inflexible, but would also make the programs illegible. You might be tempted to ignore these exceptional events and just hope they won't occur, but Python offers a powerful alternative through its exception objects.

In this chapter, you learn how to create and raise your own exceptions, as well as how to handle exceptions in various ways.

What Is an Exception?

To represent exceptional conditions, Python uses *exception objects*. When it encounters an error, it *raises* an exception. If such an exception object is not handled (or *caught*), the program terminates with a so-called *traceback* (an error message):

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

If such error messages were all you could use exceptions for, they wouldn't be very interesting. The fact is, however, that each exception is an instance of some class (in this case `ZeroDivisionError`), and these instances may be raised and caught in various ways, allowing you to trap the error and do something about it instead of just letting the entire program fail.

Making Things Go Wrong . . . Your Way

As you've seen, exceptions are raised automatically when something is wrong. Before looking at how to deal with those exceptions, let's take a look at how you can raise exceptions yourself—and even create your own kinds of exceptions.

The raise Statement

To raise an exception, you use the `raise` statement with an argument that is either a class (which should subclass `Exception`) or an instance. When using a class, an instance is created automatically Here is an example, using the built-in exception class `Exception`:

```
>>> raise Exception
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
Exception
>>> raise Exception('hyperdrive overload')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
Exception: hyperdrive overload
```

The first example, `raise Exception`, raises a generic exception with no information about what went wrong. In the last example, I added the error message `hyperdrive overload`.

Many built-in classes are available. You can find a description of all of them in the Python Library Reference, in the section "Built-in Exceptions." You can also explore them yourself with the interactive interpreter. You can find all the built-in exceptions in the module `exceptions` (as well as in the built-in namespace). To list the contents of a module, you can use the `dir` function, which is described in Chapter 10:

```
>>> import exceptions
>>> dir(exceptions)
['ArithmeticError', 'AssertionError', 'AttributeError', ...]
```

In your interpreter, this list will be quite a lot longer; I've deleted most of the names in the interest of brevity. All of these exception classes can be used in your `raise` statements:

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

Table 8-1 describes some of the most important built-in exception classes.

Table 8-1. *Some Built-in Exceptions*

Class Name	Description
`Exception`	The base class for all exceptions
`AttributeError`	Raised when attribute reference or assignment fails
`IOError`	Raised when trying to open a nonexistent file (among other things)
`IndexError`	Raised when using a nonexistent index on a sequence
`KeyError`	Raised when using a nonexistent key on a mapping
`NameError`	Raised when a name (variable) is not found

Class Name	Description
`SyntaxError`	Raised when the code is ill-formed
`TypeError`	Raised when a built-in operation or function is applied to an object of the wrong type
`ValueError`	Raised when a built-in operation or function is applied to an object with the correct type, but with an inappropriate value
`ZeroDivisionError`	Raised when the second argument of a division or modulo operation is zero

Custom Exception Classes

Although the built-in exceptions cover a lot of ground and are sufficient for many purposes, there are times when you might want to create your own. For example, in the `hyperdrive overload` example, wouldn't it be more natural to have a specific `HyperdriveError` class representing error conditions in the hyperdrive? It might seem that the error message is sufficient, but as you will see in the next section ("Catching Exceptions"), you can selectively handle certain types of exceptions based on their class. Thus, if you wanted to handle hyperdrive errors with special error-handling code, you would need a separate class for the exceptions.

So, how do you create exception classes? Just like any other class—but be sure to subclass `Exception` (either directly or indirectly, which means that subclassing any other built-in exception is okay). Thus, writing a custom exception basically amounts to something like this:

```
class SomeCustomException(Exception): pass
```

Really not much work, is it? (If you want, you can certainly add methods to your exception class as well.)

Catching Exceptions

As mentioned earlier, the interesting thing about exceptions is that you can handle them (often called *trapping* or *catching* the exceptions). You do this with the `try/except` statement. Let's say you have created a program that lets the user enter two numbers and then divides one by the other, like this:

```
x = input('Enter the first number: ')
y = input('Enter the second number: ')
print x/y
```

This would work nicely until the user enters zero as the second number:

```
Enter the first number: 10
Enter the second number: 0
Traceback (most recent call last):
  File "exceptions.py", line 3, in ?
    print x/y
ZeroDivisionError: integer division or modulo by zero
```

To catch the exception and perform some error handling (in this case simply printing a more user-friendly error message), you could rewrite the program like this:

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except ZeroDivisionError:
    print "The second number can't be zero!"
```

It might seem that a simple `if` statement checking the value of `y` would be easier to use, and in this case, it might indeed be a better solution. But if you added more divisions to your program, you would need one `if` statement per division; by using `try/except`, you need only one error handler.

■ **Note** Exceptions propagate out of functions to where they're called, and if they're not caught there either, the exceptions will "bubble up" to the top level of the program. This means that you can use `try/except` to catch exceptions that are raised in other people's functions. For more details, see the section "Exceptions and Functions," later in this chapter.

Look, Ma, No Arguments!

If you have caught an exception but you want to raise it again (pass it on, so to speak), you can call `raise` without any arguments. (You can also supply the exception explicitly if you catch it, as explained in the section "Catching the Object," later in this chapter.)

As an example of how this might be useful, consider a calculator class that has the capability to "muffle" `ZeroDivisionError` exceptions. If this behavior is turned on, the calculator prints out an error message instead of letting the exception propagate. This is useful if the calculator is used in an interactive session with a user, but if it is used internally in a program, raising an exception would be better. Therefore, the muffling can be turned off. Here is the code for such a class:

```
class MuffledCalculator:
    muffled = False
    def calc(self, expr):
        try:
            return eval(expr)
        except ZeroDivisionError:
            if self.muffled:
                print 'Division by zero is illegal'
            else:
                raise
```

■ **Note** If division by zero occurs and muffling is turned on, the `calc` method will (implicitly) return `None`. In other words, if you turn on muffling, you should not rely on the return value.

The following is an example of how this class may be used, both with and without muffling:

```
>>> calculator = MuffledCalculator()
>>> calculator.calc('10/2')
5
>>> calculator.calc('10/0') # No muffling
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "MuffledCalculator.py", line 6, in calc
    return eval(expr)
  File "<string>", line 0, in ?
ZeroDivisionError: integer division or modulo by zero
>>> calculator.muffled = True
>>> calculator.calc('10/0')
Division by zero is illegal
```

As you can see, when the calculator is not muffled, the `ZeroDivisionError` is caught but passed on.

More Than One except Clause

If you run the program from the previous section again and enter a nonnumeric value at the prompt, another exception occurs:

```
Enter the first number: 10
Enter the second number: "Hello, world!"
Traceback (most recent call last):
  File "exceptions.py", line 4, in ?
    print x/y
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Because the `except` clause looked for only `ZeroDivisionError` exceptions, this one slipped through and halted the program. To catch this exception as well, you can simply add another `except` clause to the same `try/except` statement:

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except ZeroDivisionError:
    print "The second number can't be zero!"
except TypeError:
    print "That wasn't a number, was it?"
```

This time using an `if` statement would be more difficult. How do you check whether a value can be used in division? There are a number of ways, but by far the best way is, in fact, to simply divide the values to see if it works.

Also notice how the exception handling doesn't clutter the original code. Adding a lot of `if` statements to check for possible error conditions could easily have made the code quite unreadable.

Catching Two Exceptions with One Block

If you want to catch more than one exception type with one block, you can specify them all in a tuple, as follows:

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')

    print x/y
except (ZeroDivisionError, TypeError, NameError):
    print 'Your numbers were bogus...'
```

In the preceding code, if the user either enters a string or something other than a number, or if the second number is zero, the same error message is printed. Simply printing an error message isn't very helpful, of course. An alternative could be to keep asking for numbers until the division works. I show you how to do that in the section "When All Is Well," later in this chapter.

Note that the parentheses around the exceptions in the `except` clause are important. A common error is to omit these parentheses, in which case you may end up with something other than what you want. For an explanation, see the next section, "Catching the Object."

Catching the Object

If you want access to the exception object itself in an `except` clause, you can use two arguments instead of one. (Note that even when you are catching multiple exceptions, you are supplying `except` with only one argument—a tuple.) This can be useful (for example) if you want your program to keep running, but you want to log the error somehow (perhaps just printing it out to the user). The following is a sample program that prints out the exception (if it occurs), but keeps running:

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except (ZeroDivisionError, TypeError), e:
    print e
```

■ **Note** In Python 3.0, the `except` clause will be written `except (ZeroDivisionError, TypeError) as e`.

The except clause in this little program again catches two types of exceptions, but because you also explicitly catch the object itself, you can print it out so the user can see what happened. (You see a more useful application of this later in this chapter, in the section "When All Is Well.")

A Real Catchall

Even if the program handles several types of exceptions, some may still slip through. For example, using the same division program, simply try to press Enter at the prompt, without writing anything. You should get an error message and some information about what went wrong (a *stack trace*), somewhat like this:

```
Traceback (most recent call last):
  File 'exceptions.py', line 3, in ?
    x = input('Enter the first number: ')
  File '<string>', line 0

    ^
SyntaxError: unexpected EOF while parsing
```

This exception got through the try/except statement—and rightly so. You hadn't foreseen that this could happen and weren't prepared for it. In these cases, it is better that the program crash immediately (so you can see what's wrong) than that it simply hide the exception with a try/except statement that isn't meant to catch it.

However, if you *do* want to catch *all* exceptions in a piece of code, you can simply omit the exception class from the except clause:

```
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except:
    print 'Something wrong happened...'
```

Now you can do practically whatever you want:

```
Enter the first number: "This" is *completely* illegal 123
Something wrong happened...
```

■ **Caution** Catching all exceptions like this is risky business because it will hide errors you haven't thought of as well as those you're prepared for. It will also trap attempts by the user to terminate execution by Ctrl-C, attempts by functions you call to terminate by sys.exit, and so on. In most cases, it would be better to use except Exception, e and perhaps do some checking on the exception object, e.

When All Is Well

In some cases, it can be useful to have a block of code that is executed *unless* something bad happens; as with conditionals and loops, you can add an `else` clause to the `try/except` statement:

```
try:
    print 'A simple task'
except:
    print 'What? Something went wrong?'
else:
    print 'Ah... It went as planned.'
```

If you run this, you get the following output:

```
A simple task
Ah... It went as planned.
```

With this `else` clause, you can implement the loop hinted at in the section "Catching Two Exceptions with One Block," earlier in this chapter:

```
while True:
    try:
        x = input('Enter the first number: ')
        y = input('Enter the second number: ')
        value = x/y
        print 'x/y is', value
    except:
        print 'Invalid input. Please try again.'
    else:
        break
```

Here, the loop is broken (by the `break` statement in the `else` clause) only when no exception is raised. In other words, as long as something wrong happens, the program keeps asking for new input. The following is an example run:

```
Enter the first number: 1
Enter the second number: 0
Invalid input. Please try again.
Enter the first number: 'foo'
Enter the second number: 'bar'
Invalid input. Please try again.
Enter the first number: baz
Invalid input. Please try again.
Enter the first number: 10
Enter the second number: 2
x/y is 5
```

As mentioned previously, an alternative to using an empty `except` clause is to catch all exceptions of the `Exception` class (which will catch all exceptions of any subclass as well). You cannot be 100 percent certain that you'll catch everything then, because the code in your `try/except` statement may be naughty and use the old-fashioned string exceptions, or perhaps create a custom exception that doesn't subclass `Exception`. However, if you go with the `except Exception` version, you can use the technique from the section "Catching the Object," earlier in this chapter, to print out a more instructive error message in your little division program:

```
while True:
    try:
        x = input('Enter the first number: ')
        y = input('Enter the second number: ')
        value = x/y
        print 'x/y is', value
    except Exception, e:
        print 'Invalid input:', e
        print 'Please try again'
    else:
        break
```

The following is a sample run:

```
Enter the first number: 1
Enter the second number: 0
Invalid input: integer division or modulo by zero
Please try again
Enter the first number: 'x'
Enter the second number: 'y'
Invalid input: unsupported operand type(s) for /: 'str' and 'str'
Please try again
Enter the first number: quuux
Invalid input: name 'quuux' is not defined
Please try again
Enter the first number: 10
Enter the second number: 2
x/y is 5
```

And Finally . . .

Finally, there is the `finally` clause. You use it to do housekeeping after a possible exception. It is combined with a `try` clause:

```
x = None
try:
    x = 1/0
```

```
finally:
    print 'Cleaning up...'
    del x
```

In the preceding example, you are *guaranteed* that the `finally` clause will be executed, no matter what exceptions occur in the `try` clause. The reason for initializing `x` before the `try` clause is that otherwise it would never be assigned a value because of the `ZeroDivisionError`. This would lead to an exception when using `del` on it within the `finally` clause, which you *wouldn't* catch.

If you run this, the cleanup comes *before* the program crashes and burns:

```
Cleaning up...
Traceback (most recent call last):
  File "C:\python\div.py", line 4, in ?
    x = 1/0
ZeroDivisionError: integer division or modulo by zero
```

While using `del` to remove a variable is a rather silly kind of cleanup, the `finally` clause may be quite useful for closing files or network sockets and the like. (More on those in Chapter 14.)

You can also combine `try`, `except`, `finally`, and `else` (or just three of them) in a single statement:

```
try:
    1/0
except NameError:
    print "Unknown variable"
else:
    print "That went well!"
finally:
    print "Cleaning up."
```

■ **Note** In Python versions prior to 2.5, the `finally` clause had to be used on its own—it couldn't be used in the same `try` statement as an `except` clause. If you wanted both, you needed to wrap two statements. From Python 2.5 onwards, you can combine these to your heart's content, though.

Exceptions and Functions

Exceptions and functions work together quite naturally. If an exception is raised inside a function, and isn't handled there, it propagates (*bubbles up*) to the place where the function was called. If it isn't handled there either, it continues propagating until it reaches the main program (the global scope), and if there is no exception handler there, the program halts with a stack trace. Let's take a look at an example:

```
>>> def faulty():
...     raise Exception('Something is wrong')
...
```

```
>>> def ignore_exception():
...     faulty()
...

>>> def handle_exception():
...     try:
...         faulty()
...     except:
...         print 'Exception handled'
...
>>> ignore_exception()
Traceback (most recent call last):
  File '<stdin>', line 1, in ?
  File '<stdin>', line 2, in ignore_exception
  File '<stdin>', line 2, in faulty
Exception: Something is wrong
>>> handle_exception()
Exception handled
```

As you can see, the exception raised in `faulty` propagates through `faulty` and `ignore_exception`, and finally causes a stack trace. Similarly, it propagates through to `handle_exception`, but there it is handled with a `try/except` statement.

The Zen of Exceptions

Exception handling isn't very complicated. If you know that some part of your code may cause a certain kind of exception, and you don't simply want your program to terminate with a stack trace if and when that happens, then you add the necessary `try/except` or `try/finally` statements (or some combination thereof) to deal with it, as needed.

Sometimes, you can accomplish the same thing with conditional statements as you can with exception handling, but the conditional statements will probably end up being less natural and less readable. On the other hand, some things that might seem like natural applications of `if/else` may in fact be implemented much better with `try/except`. Let's take a look at a couple of examples.

Let's say you have a dictionary and you want to print the value stored under a specific key, if it is there. If it isn't there, you don't want to do anything. The code might be something like this:

```
def describePerson(person):
    print 'Description of', person['name']
    print 'Age:', person['age']
    if 'occupation' in person:
        print 'Occupation:', person['occupation']
```

If you supply this function with a dictionary containing the name Throatwobbler Mangrove and the age 42 (but no occupation), you get the following output:

```
Description of Throatwobbler Mangrove
Age: 42
```

If you add the occupation "camper," you get the following output:

```
Description of Throatwobbler Mangrove
Age: 42
Occupation: camper
```

The code is intuitive, but a bit inefficient (although the main concern here is really code simplicity). It has to look up the key `'occupation'` twice—once to see whether the key exists (in the condition) and once to get the value (to print it out). An alternative definition is as follows:

```
def describePerson(person):
    print 'Description of', person['name']
    print 'Age:', person['age']
    try:
        print 'Occupation: ' + person['occupation']
    except KeyError: pass
```

■ **Note** I use + instead of a comma for printing the occupation here; otherwise, the string `'Occupation:'` would have been printed before the exception is raised.

Here, the function simply assumes that the key `'occupation'` is present. If you assume that it normally is, this saves some effort. The value will be fetched and printed—no extra fetch to check whether it is indeed there. If the key doesn't exist, a `KeyError` exception is raised, which is trapped by the `except` clause.

You may also find `try/except` useful when checking whether an object has a specific attribute. Let's say you want to check whether an object has a `write` attribute, for example. Then you could use code like this:

```
try:
    obj.write
except AttributeError:
    print 'The object is not writeable'
else:
    print 'The object is writeable'
```

Here the `try` clause simply accesses the attribute without doing anything useful with it. If an `AttributeError` is raised, the object doesn't have the attribute; otherwise, it has the

attribute. This is a natural alternative to the `getattr` solution introduced in Chapter 7 (in the section "Interfaces and Introspection"). Which one you prefer is largely a matter of taste. Indeed, `getattr` is internally implemented in exactly this way: it tries to access the attribute and catches the `AttributeError` that this attempt may raise.

Note that the gain in efficiency here isn't great. (It's more like really, really tiny.) In general (unless your program is having performance problems), you shouldn't worry about that sort of optimization too much. The point is that using `try/except` statements is in many cases much more natural (more "Pythonic") than `if/else`, and you should get into the habit of using them where you can.[1]

A Quick Summary

The main topics covered in this chapter are as follows:

Exception objects: Exceptional situations (such as when an error has occurred) are represented by exception objects. These can be manipulated in several ways, but if ignored, they terminate your program.

Warnings: Warnings are similar to exceptions, but will (in general) just print out an error message.

Raising exceptions: You can raise exceptions with the `raise` statement. It accepts either an exception class or an exception instance as its argument. You can also supply two arguments (an exception and an error message). If you call `raise` with no arguments in an `except` clause, it "reraises" the exception caught by that clause.

Custom exception classes: You can create your own kinds of exceptions by subclassing `Exception`.

Catching exceptions: You catch exceptions with the `except` clause of a `try` statement. If you don't specify a class in the `except` clause, all exceptions are caught. You can specify more than one class by putting them in a tuple. If you give two arguments to `except`, the second is bound to the exception object. You can have several `except` clauses in the same `try/except` statement, to react differently to different exceptions.

`else` **clauses**: You can use an `else` clause in addition to `except`. The `else` clause is executed if no exceptions are raised in the main `try` block.

`finally`: You can use `try/finally` if you need to make sure that some code (for example, cleanup code) is executed, regardless of whether or not an exception is raised. This code is then put in the `finally` clause.

Exceptions and functions: When you raise an exception inside a function, it propagates to the place where the function was called. (The same goes for methods.)

1. The preference for `try/except` in Python is often explained through Rear Admiral Grace Hopper's words of wisdom, "It's easier to ask forgiveness than permission." This strategy of simply trying to do something and dealing with any errors, rather than doing a lot of checking up front, is called the *Leap Before You Look* idiom.

New Functions in This Chapter

Function	Description
`warnings.filterwarnings(action, ...)`	Used to filter out warnings

What Now?

While you might think that the material in this chapter was exceptional (pardon the pun), the next chapter is truly magical. Well, *almost* magical.