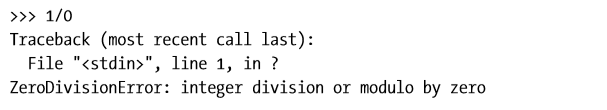
**Python Exceptions**

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 events, you might use conditionals every-where 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 program illegible. You might be tempted to ignore these exceptions and just hope they won’t occur, but Python offers a powerful alternative.

**Please note: Because we are using the latest version of Python, when you type in your code in your ide/interpreter, replace print “” with print(“”). The contents of this handout was adapted from a book which still uses the old style of printing.**

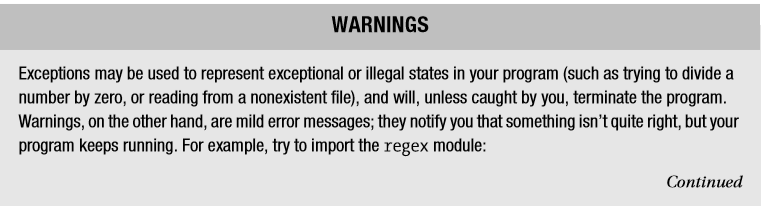
**What Is an Exception?**

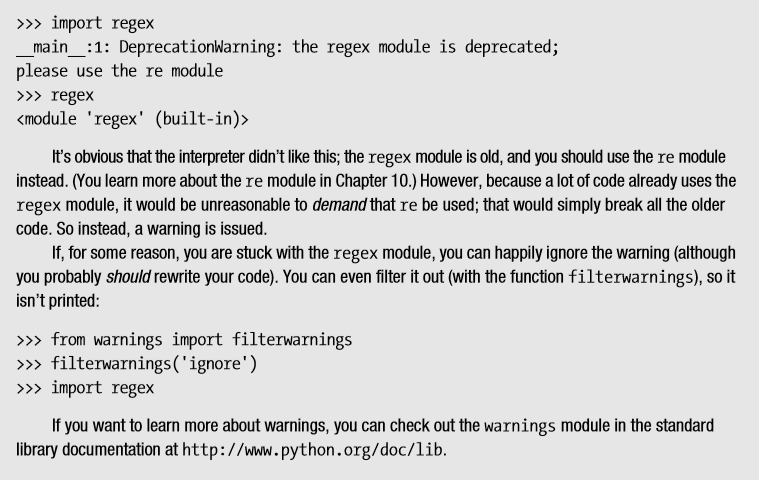
To represent exceptional conditions, Python uses exception objects(exception objects represent an error). If such an exception object is not handled in any way, the program terminates with so-called traceback (an error message) An exception are errors detected during execution of a program:



If such error messages were all you could use exceptions for, exceptions 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 allowing the entire program to fail.

In this section, you learn how to create and raise your own exceptions. In the following sections, you learn about handling exceptions in various ways.

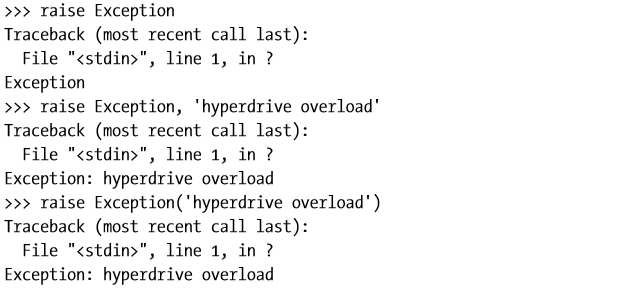




**Making Things Go Wrong**

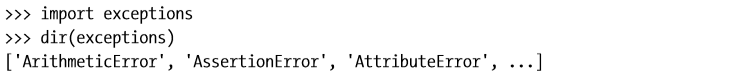
As you’ve seen, exceptions are raised automatically when something is wrong. Before looking at how to deal with those exceptions, let’s take a look at how you can raise exceptions yourself and even create your own kinds of exceptions.

**The raise Statement**

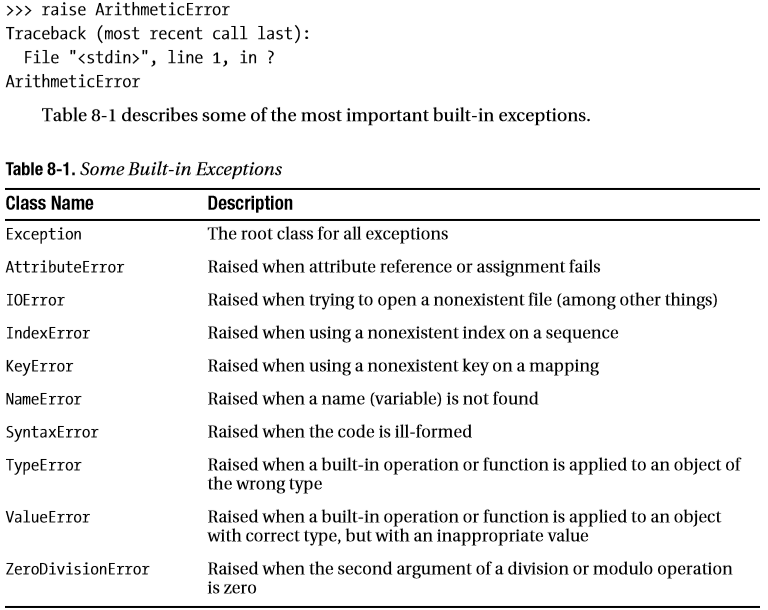


The first example (**raise Exception**) raises a generic exception with no information of what went wrong. In the last two examples, I added the error message *hyperdrive overload*. As you can see, the two forms *raise class, messag*e and *raise class(message)* are equivalent; both raise an exception with the given error message.

There are many built-in classes 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; they are all found in the module exceptions, for your convenience (as well as in the built-in namespace). To list the contents of a module, you can use the **dir** function that we learnt earlier.

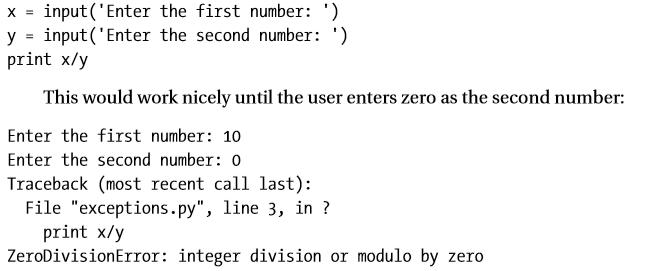


In your interpreter, this list will be quite a lot longer—I’ve deleted most of the names in the interest of legibility. All of these exceptions can be used in your raise statements:

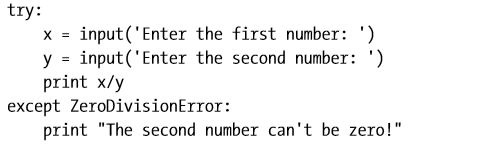


**Catching Exceptions**

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:



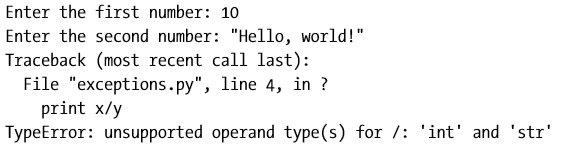
To catch the exception and perform some error handling (in this case simply printing a more user-friendly error message), you could rewrite the program like this:



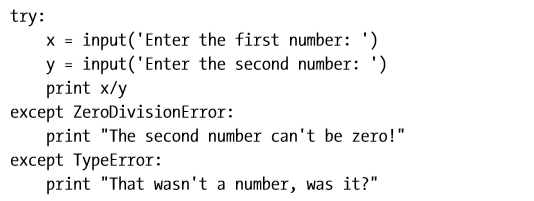
It might seem that a simple if statement checking the value of y would be easier to use, and in this case it might indeed be a better solution. But if you added more divisions to your program, you would need one if statement per division; by using *try/except*, you need only one error handler.

**More Than One except Clause**

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



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

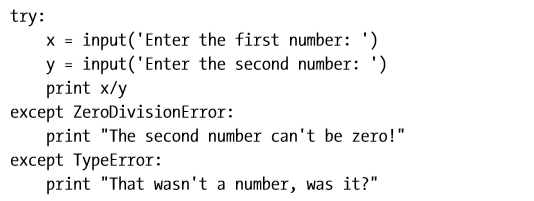


This time using an if statement would be more difficult. How do you check whether a value can be used in division? There are a number of ways, but by far the best way is, in fact, to simply divide the values to see if it works.

Also notice how the exception handling doesn’t clutter the original code; adding lots 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:

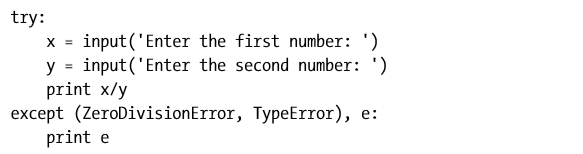


In the preceding code, if the user either enters a string or something other than a number, or if the second number is zero, the same error message is printed. Simply printing an error message isn’t very helpful, of course. An alternative could be to keep asking for numbers until the division works.

Note that the parentheses around the exceptions in the except clause are important; a common error is to omit them, 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**

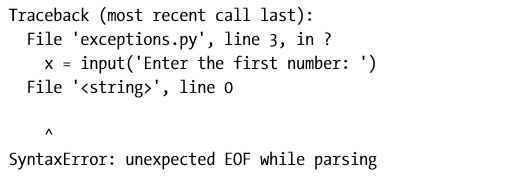
If you want access to the exception itself in an except clause, you can use two arguments instead of one. (Note that even when you are catching multiple exceptions, you are only supplying except with one argument—a tuple.) This can be useful (for example) if you want your program to keep running, but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



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.”)

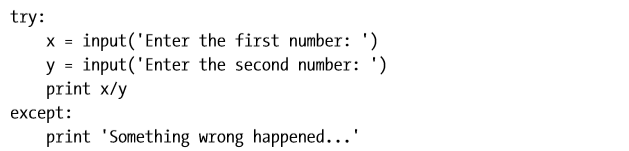
**A Real Catchall**

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 a stack trace somewhat like this:



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 crashes immediately (so you can see what’s wrong) than that it simply hides 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:



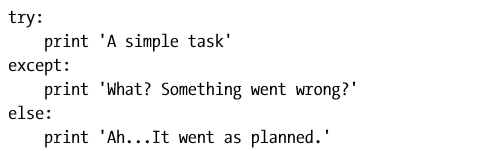
Now you can practically do whatever you want:



|  |
| --- |
| **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**

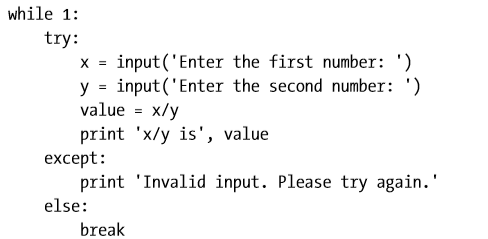
In some cases, it can be useful to have a block of code that is executed unless something bad happens; as with conditionals and loops, you can add an else clause:



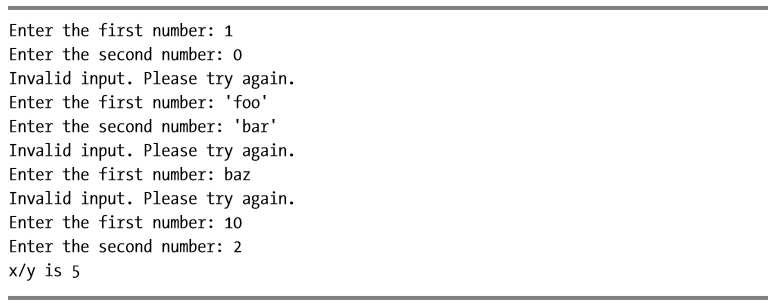
If you run this, you get the following output:



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



Here the loop is only broken (by the **break** statement in the else clause) 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:



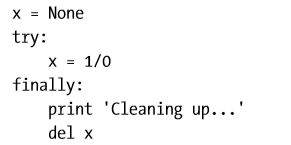
As mentioned previously, an alternative to using an empty except clause is to catch all exceptions of the Exception class (which will catch all exceptions of any subclass as well).

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.

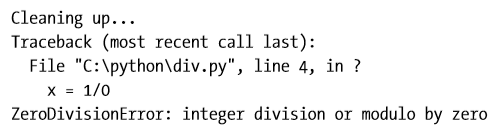
However, if you go with the except Exception version, you can use the technique from the section “Catching the Object,” earlier in this chapter, to print out a more instructive error message in your little division program.

**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 (**but not an except clause**):



In the preceding, you are guaranteed that the finally clause will get 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 get 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:



**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 an exception is raised or not. This code is then put in the finally clause. Note that you cannot have both except clauses and a finally clause in the same try statement—but you can put one inside the other