Chapter 9

Dealing with Errors

In This Chapter

- Defining problems in communication with Python
- Understanding error sources
- Handling error conditions
- Specifying that an error has occurred
- Developing your own error indicators
- Performing tasks even after an error occurs

Most application code of any complexity has errors in it. When your application suddenly freezes for no apparent reason, that's an error. Seeing one of those obscure message dialog boxes is another kind of error. However, errors can occur that don't provide you with any sort of notification. An application might perform the wrong computation on a series of numbers you provide, resulting in incorrect output that you may never know about unless someone tells you that something is wrong or you check for the issue yourself. Errors need not be consistent, either. You may see them on some occasions and not on others. For example, an error can occur only when the weather is bad or the network is overloaded. In short, errors occur in all sorts of situations and for all sorts of reasons. This chapter tells you about all sorts of errors and what to do when your application encounters them.

It shouldn't surprise you that errors occur — applications are written by humans, and humans make mistakes. Most developers call application errors *exceptions,* meaning that they're the exception to the rule. Because exceptions do occur in applications, you need to detect and do something about them whenever possible. The act of detecting and processing an exception is called *error handling* or *exception handling.* In order to properly detect errors, you need to know about error sources and why errors occur in the first place. When you do detect the error, you must process it by *catching* the exception. Catching an exception means examining it and possibly doing something about it. So, another part of this chapter is about discovering how to perform exception handling in your own application.

Sometimes your code detects an error in the application. When this happens, you need to *raise* or *throw* an exception. You see both terms used for the same thing, which simply means that your code encountered an error it couldn't handle, so it passed the error information onto another piece of code to *handle* (interpret, process, and, with luck, fix the exception). In some cases, you use custom error message objects to pass on the information. Even though Python has a wealth of generic message objects that cover most situations, some situations are special. For example, you might want to provide special support for a database application, and Python won't normally cover that contingency with a generic message object. It's important to know when to handle exceptions locally, when to send them to the code that called your code, and when to create special exceptions so that every part of the application knows how to handle the exception — all topics covered by this chapter.

There are also times when you must ensure that your application handles an exception gracefully, even if that means shutting the application down. Fortunately, Python provides the `finally` clause, which always executes, even when an exception occurs. You can place code to close files or perform other essential tasks in the code block associated with this clause. Even though you won't perform this task all the time, it's the last topic discussed in the chapter.

Knowing Why Python Doesn't Understand You

Developers often get frustrated with programming languages and computers because they seemingly go out of their way to cause communication problems. Of course, programming languages and computers are both inanimate — there is no desire for anything on the part of either. Programming languages and computers also don't think; they accept whatever the developer has to say quite literally. Therein lies the problem.

Neither Python nor the computer will "know what you mean" when you type instructions as code. Both follow whatever instructions you provide to the letter and literally as you provide them. You may not have meant to tell Python to delete a data file unless some absurd condition occurred. However, if you don't make the conditions clear, Python will delete the file whether the condition exists or not. When an error of this sort happens, people commonly say that the application has a *bug* in it. Bugs are simply coding errors that you can remove using a debugger. (A *debugger* is a special kind of tool that lets you stop or pause application execution, examine the content of variables, and generally dissect the application to see what makes it tick.)

Errors occur in many cases when the developer makes assumptions that simply aren't true. Of course, this includes assumptions about the application user, who probably doesn't care about the extreme level of care you took when crafting your application. The user will enter bad data. Again, Python won't know or care that the data is bad and will process it even when your intent was to disallow the bad input. Python doesn't understand the concepts of good or bad data; it simply processes incoming data according to any rules you set, which means that you must set rules to protect users from themselves.

Python isn't proactive or creative — those qualities exist only in the developer. When a network error occurs or the user does something unexpected, Python doesn't create a solution to fix the problem. It only processes code. If you don't provide code to handle the error, the application is likely to fail and crash ungracefully — possibly taking all of the user's data with it. Of course, the developer can't anticipate every potential error situation, either, which is why most complex applications have errors in them — errors of omission, in this case.

Some developers out there think they can create bulletproof code, despite the absurdity of thinking that such code is even possible. Smart developers assume that some number of bugs will get through the code-screening process, that nature and users will continue to perform unexpected actions, and that even the smartest developer can't anticipate every possible error condition. Always assume that your application is subject to errors that will cause exceptions; that way, you'll have the mindset required to actually make your application more reliable.

Considering the Sources of Errors

You might be able to divine the potential sources of error in your application by reading tea leaves, but that's hardly an efficient way to do things. Errors actually fall into well-defined categories that help you predict (to some degree) when and where they'll occur. By thinking about these categories as you work through your application, you're far more likely to discover potential errors sources before they occur and cause potential damage. The two principle categories are

- Errors that occur at a specific time
- Errors that are of a specific type

The following sections discuss these two categories in greater detail. The overall concept is that you need to think about error classifications in order to start finding and fixing potential errors in your application before they become a problem.

Classifying when errors occur

Errors occur at specific times. The two major time frames are

- Compile time
- Runtime

No matter when an error occurs, it causes your application to misbehave. The following sections describe each time frame.

Compile time

A compile time error occurs when you ask Python to run the application. Before Python can run the application, it must interpret the code and put it into a form that the computer can understand. A computer relies on machine code that is specific to that processor and architecture. If the instructions you write are malformed or lack needed information, Python can't perform the required conversion. It presents an error that you must fix before the application can run.

Fortunately, compile-time errors are the easiest to spot and fix. Because the application won't run with a compile-time error in place, user never sees this error category. You fix this sort of error as you write your code.

The appearance of a compile-time error should tell you that other typos or omissions could exist in the code. It always pays to check the surrounding code to ensure that no other potential problems exist that might not show up as part of the compile cycle.

Runtime

A runtime error occurs after Python compiles the code you write and the computer begins to execute it. Runtime errors come in several different types, and some are harder to find than others. You know you have a runtime error when the application suddenly stops running and displays an exception dialog box or when the user complains about erroneous output (or at least instability).

Not all runtime errors produce an exception. Some runtime errors cause instability (the application freezes), errant output, or data damage. Runtime errors can affect other applications or create unforeseen damage to the platform on which the application is running. In short, runtime errors can cause you quite a bit of grief, depending on precisely the kind of error you're dealing with at the time.

Many runtime errors are caused by errant code. For example, you can misspell the name of a variable, preventing Python from placing information in the correct variable during execution. Leaving out an optional but necessary

argument when calling a method can also cause problems. These are examples of *errors of commission,* which are specific errors associated with your code. In general, you can find these kinds of errors using a debugger or by simply reading your code line by line to check for errors.

Runtime errors can also be caused by external sources not associated with your code. For example, the user can input incorrect information that the application isn't expecting, causing an exception. A network error can make a required resource inaccessible. Sometimes even the computer hardware has a glitch that causes a nonrepeatable application error. These are all examples of *errors of omission,* from which the application might recover if your application has error-trapping code in place. It's important that you consider both kinds of runtime errors — errors of commission and omission — when building your application.

Distinguishing error types

You can distinguish errors by type, that is, by how they're made. Knowing the error types helps you understand where to look in an application for potential problems. Exceptions work like many other things in life. For example, you know that electronic devices don't work without power. So, when you try to turn your television on and it doesn't do anything, you might look to ensure that the power cord is firmly seated in the socket.

Understanding the error types helps you locate errors faster, earlier, and more consistently, resulting in fewer misdiagnoses. The best developers know that fixing errors while an application is in development is always easier than fixing it when the application is in production because users are inherently impatient and want errors fixed immediately and correctly. In addition, fixing an error earlier in the development cycle is always easier than fixing it when the application nears completion because less code exists to review.

The trick is to know where to look. With this in mind, Python (and most other programming languages) breaks errors into the following types:

- Syntactical
- Semantic
- Logical

The following sections examine each of these error types in more detail. I've arranged the sections in order of difficulty, starting with the easiest to find. A syntactical error is generally the easiest; a logical error is generally the hardest.

Syntactical

Whenever you make a typo of some sort, you create a syntactical error. Some Python syntactical errors are quite easy to find because the application simply doesn't run. The interpreter may even point out the error for you by highlighting the errant code and displaying an error message. However, some syntactical errors are quite hard to find. Python is case sensitive, so you may use the wrong case for a variable in one place and find that the variable isn't quite working as you thought it would. Finding the one place where you used the wrong capitalization can be quite challenging.

Most syntactical errors occur at compile time and the interpreter points them out for you. Fixing the error is made easy because the interpreter generally tells you what to fix, and with considerable accuracy. Even when the interpreter doesn't find the problem, syntactical errors prevent the application from running correctly, so any errors the interpreter doesn't find show up during the testing phase. Few syntactical errors should make it into production as long as you perform adequate application testing.

Semantic

When you create a loop that executes one too many times, you don't generally receive any sort of error information from the application. The application will happily run because it thinks that it's doing everything correctly, but that one additional loop can cause all sorts of data errors. When you create an error of this sort in your code, it's called a *semantic error.*

Semantic errors occur because the meaning behind a series of steps used to perform a task is wrong — the result is incorrect even though the code apparently runs precisely as it should. Semantic errors are tough to find, and you sometimes need some sort of debugger to find them. (Chapter 19 provides a discussion of tools that you can use with Python to perform tasks such as debugging applications. You can also find blog posts about debugging on my blog at `http://blog.johnmuellerbooks.com`.)

Logical

Some developers don't create a division between semantic and logical errors, but they are different. A semantic error occurs when the code is essentially correct but the implementation is wrong (such as having a loop execute once too often). Logical errors occur when the developer's thinking is faulty. In many cases, this sort of error happens when the developer uses a relational or logical operator incorrectly. However, logical errors can happen in all sorts of other ways, too. For example, a developer might think that data is always stored on the local hard drive, which means that the application may behave in an unusual manner when it attempts to load data from a network drive instead.

Logical errors are quite hard to fix because the problem isn't with the actual code, yet the code itself is incorrectly defined. The thought process that went into creating the code is faulty; therefore, the developer who created the error is less likely to find it. Smart developers use a second pair of eyes to help spot logical errors. Having a formal application specification also helps because the logic behind the tasks the application performs is usually given a formal review.

Catching Exceptions

Generally speaking, a user should never see an exception dialog box. Your application should always catch the exception and handle it before the user sees it. Of course, the real world is different — users do see unexpected exceptions from time to time. However, catching every potential exception is still the goal when developing an application. The following sections describe how to catch exceptions and handle them.

Understanding the built-in exceptions

Python comes with a host of built-in exceptions — far more than you might think possible. You can see a list of these exceptions at `https://docs.python.org/3.3/library/exceptions.html`. The documentation breaks the exception list down into categories. Here is a brief overview of the Python exception categories that you work with regularly:

- **Base classes:** The base classes provide the essential building blocks (such as the `Exception` exception) for other exceptions. However, you might actually see some of these exceptions, such as the `ArithmeticError` exception, when working with an application.
- **Concrete exceptions:** Applications can experience hard errors — errors that are hard to overcome because there really isn't a good way to handle them or they signal an event that the application must handle. For example, when a system runs out of memory, Python generates a `MemoryError` exception. Recovering from this error is hard because it isn't always possible to release memory from other uses. When the user presses an interrupt key (such as Ctrl+C or Delete), Python generates a `KeyboardInterrupt` exception. The application must handle this exception before proceeding with any other tasks.
- **OS exceptions:** The operating system can generate errors that Python then passes them along to your application. For example, if your application tries to open a file that doesn't exist, the operating system generates a `FileNotFoundError` exception.
- **Warnings:** Python tries to warn you about unexpected events or actions that could result in errors later. For example, if you try to inappropriately use a resource, such as an icon, Python generates a `ResourceWarning` exception. It's important to remember that this particular category is a warning and not an actual error: Ignoring it can cause you woe later, but you can ignore it.

Basic exception handling

To handle exceptions, you must tell Python that you want to do so and then provide code to perform the handling tasks. You have a number of ways in which you can perform this task. The following sections start with the simplest method first and then move on to more complex methods that offer added flexibility.

Handling a single exception

In Chapter 7, the `IfElse.py` and other examples have a terrible habit of spitting out exceptions when the user inputs unexpected values. Part of the solution is to provide range checking. However, range checking doesn't overcome the problem of a user typing text such as Hello in place of an expected numeric value. Exception handling provides a more complex solution to the problem, as described in the following steps. This example also appears with the downloadable source code as `BasicException1.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   Value = int(input("Type a number between 1 and 10:
         "))
except ValueError:
   print("You must type a number between 1 and 10!")
else:

   if (Value > 0) and (Value <= 10):
      print("You typed: ", Value)
   else:
      print("The value you typed is incorrect!")
```

   The code within the `try` block has its exceptions handled. In this case, handling the exception means getting input from the user using the `int(input())` calls. If an exception occurs outside this block, the code doesn't handle it. With reliability in mind, the temptation might be to enclose all the executable code in a `try` block so that every exception would be handled. However, you want to make your exception handling small and specific to make locating the problem easier.

The `except` block looks for a specific exception in this case: `ValueError`. When the user creates a `ValueError` exception by typing Hello instead of a numeric value, this particular exception block is executed. If the user were to generate some other exception, this `except` block wouldn't handle it.

The `else` block contains all the code that is executed when the `try` block code is successful (doesn't generate an exception). The remainder of the code is in this block because you don't want to execute it unless the user does provide valid input. When the user provides a whole number as input, the code can then range check it to ensure that it's correct.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks you to type a number between 1 and 10.

4. **Type** Hello **and press Enter.**

   The application displays an error message, as shown in Figure 9-1.

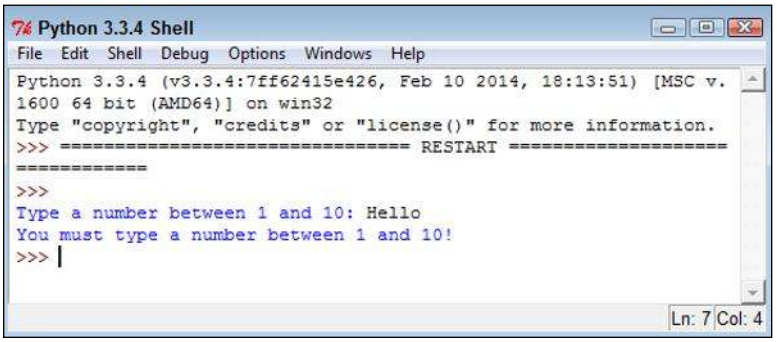

Figure 9-1: Typing the wrong input type generates an error instead of an exception.

5. **Perform Steps 3 and 4 again, but type** 5.5 **instead of Hello.**

   The application generates the same error message, as shown in Figure 9-1.

6. **Perform Steps 3 and 4 again, but type** 22 **instead of Hello.**

   The application outputs the expected range error message, as shown in Figure 9-2. Exception handling doesn't weed out range errors. You must still check for them separately.

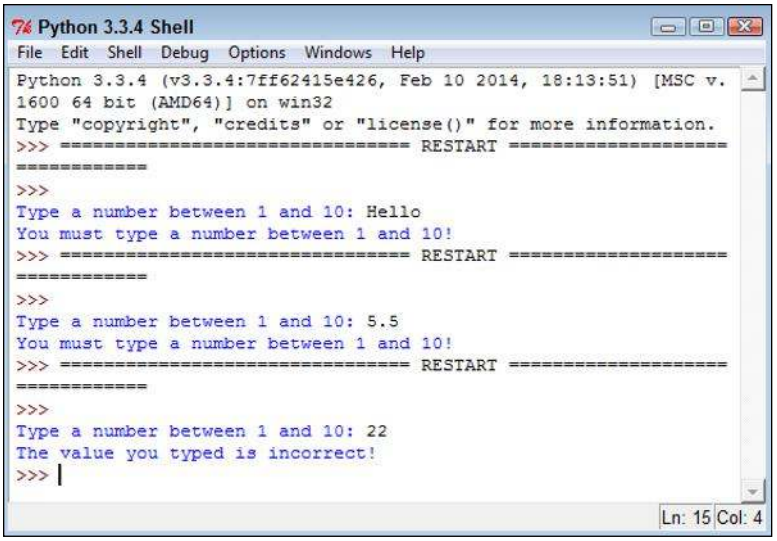

Figure 9-2: Exception handling doesn't ensure that the value is in the correct range.

7. **Perform Steps 3 and 4 again, but type** 7 **instead of Hello.**

   This time, the application finally reports that you've provided a correct value of 7. Even though it seems like a lot of work to perform this level of checking, you can't really be certain that your application is working correctly without it.

8. **Perform Steps 3 and 4 again, but press Ctrl+C, Cmd+C, or the alternative for your platform instead of typing anything.**

   The application generates a `KeyboardInterrupt` exception, as shown in Figure 9-3. Because this exception isn't handled, it's still a problem for the user. You see several techniques for fixing this problem later in the chapter.

Using the except clause without an exception

You can create an exception handling block in Python that's generic because it doesn't look for a specific exception. In most cases, you want to provide a specific exception when performing exception handling for these reasons:

- To avoid hiding an exception you didn't consider when designing the application
- To ensure that others know precisely which exceptions your application will handle
- To handle the exceptions correctly using specific code for that exception

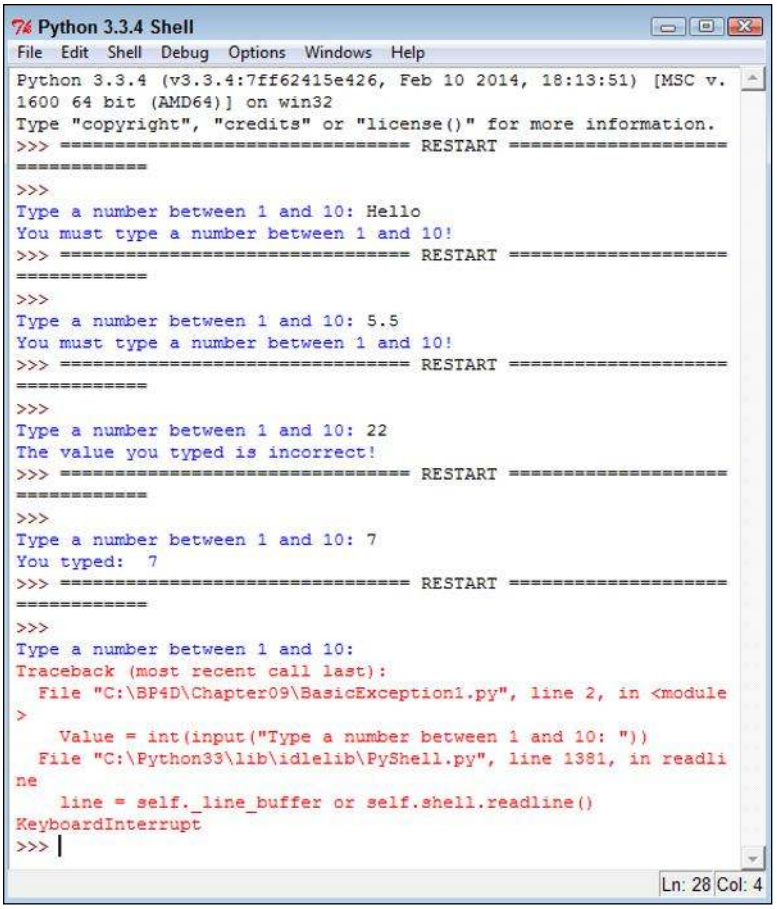

Figure 9-3: The exception handling in this example deals only with `ValueError` exceptions.

However, sometimes you may need a generic exception-handling capability, such as when you're working with third-party libraries or interacting with an external service. The following steps demonstrate how to use an `except` clause without a specific exception attached to it. This example also appears with the downloadable source code as `BasicException2.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   Value = int(input("Type a number between 1 and 10:
         "))
except:
   print("You must type a number between 1 and 10!")
else:

   if (Value > 0) and (Value <= 10):
      print("You typed: ", Value)
   else:
      print("The value you typed is incorrect!")
```

   The only difference between this example and the previous example is that the `except` clause doesn't have the `ValueError` exception specifically associated with it. The result is that this `except` clause will also catch any other exception that occurs.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks you to type a number between 1 and 10.

4. **Type** Hello **and press Enter.**

   The application displays an error message (refer to Figure 9-1).

5. **Perform Steps 3 and 4 again, but type** 5.5 **instead of Hello.**

   The application generates the same error message (again, refer to Figure 9-1).

6. **Perform Steps 3 and 4 again, but type** 22 **instead of Hello.**

   The application outputs the expected range error message (refer to Figure 9-2). Exception handling doesn't weed out range errors. You must still check for them separately.

7. **Perform Steps 3 and 4 again, but type** 7 **instead of Hello.**

   This time, the application finally reports that you've provided a correct value of 7. Even though it seems like a lot of work to perform this level of checking, you can't really be certain that your application is working correctly without it.

8. **Perform Steps 3 and 4 again, but press Ctrl+C, Cmd+C, or the alternative for your platform instead of typing anything.**

   You see the error message that's usually associated with input error, as shown in Figure 9-4. The error message is incorrect, which might confuse users. However, the plus side is that the application didn't crash,

which means that you won't lose any data and the application can recover. Using generic exception handling does have some advantages, but you must use it carefully.

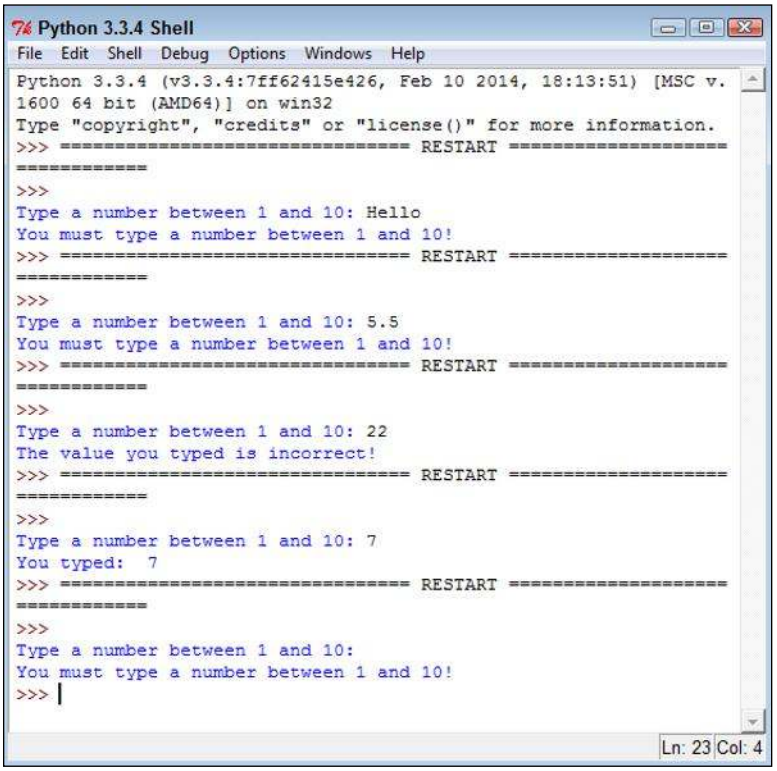

Figure 9-4: Generic exception handling traps the `KeyboardInterrupt` exception.

Working with exception arguments

Most exceptions don't provide arguments (a list of values that you can check for additional information). The exception either occurs or it doesn't. However, a few exceptions do provide arguments, and you see them used later in the book. The arguments tell you more about the exception and provide details that you need to correct it.

For the sake of completeness, this chapter includes a simple example that generates an exception with an argument. You can safely skip the remainder of this section if desired because the information is covered in more detail later in the book. This example also appears with the downloadable source code as `ExceptionWithArguments.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
import sys

try:
   File = open('myfile.txt')
except IOError as e:
   print("Error opening file!\r\n" +
      "Error Number: {0}\r\n".format(e.errno) +
      "Error Text: {0}".format(e.strerror))
else:
   print("File opened as expected.")
   File.close();
```

   This example uses some advanced features. The `import` statement obtains code from another file. Chapter 10 tells you how to use this Python feature.

   The `open()` function opens a file and provides access to the file through the `File` variable. Chapter 15 tells you how file access works. Given that `myfile.txt` doesn't exist in the application directory, the operating system can't open it and will tell Python that the file doesn't exist.

   Trying to open a nonexistent file generates an `IOError` exception. This particular exception provides access to two arguments:

   - `errno`: Provides the operating system error number as an integer
   - `strerror`: Contains the error information as a human-readable string

   The `as` clause places the exception information into a variable, `e`, that you can access as needed for additional information. The `except` block contains a `print()` call that formats the error information into an easily read error message.

   If you should decide to create the `myfile.txt` file, the `else` clause executes. In this case, you see a message stating that the file opened normally. The code then closes the file without doing anything with it.

3. **Choose Run➪Run Module.**

   You see a Python Shell window open. The application displays the Error opening file information, as shown in Figure 9-5.

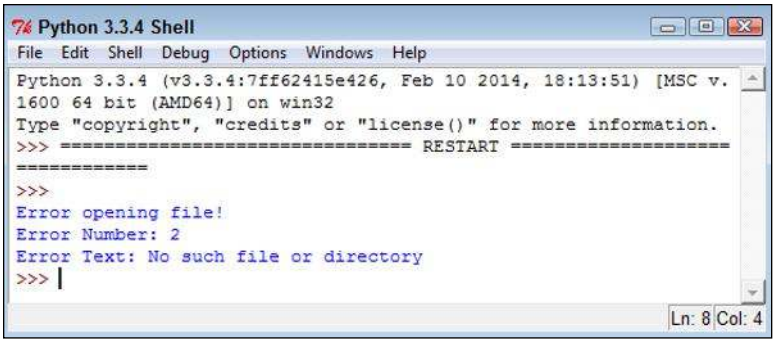

Figure 9-5: Attempting to open a nonexistent file never works.

Obtaining a list of exception arguments

The list of arguments supplied with exceptions varies by exception and by what the sender provides. It isn't always easy to figure out what you can hope to obtain in the way of additional information. One way to handle the problem is to simply print everything using code like this (this example also appears with the downloadable source code as `GetExceptionArguments1.py`):

```
import sys

try:
   File = open('myfile.txt')
except IOError as e:
   for Arg in e.args:
      print(Arg)
else:
   print("File opened as expected.")
   File.close();
```

The `args` property always contains a list of the exception arguments in string format. You can use a simple `for` loop to print each of the arguments. The only problem with this approach is that you're missing the argument names, so you know the output information (which is obvious in this case), but you don't know what to call it.

A more complex method of dealing with the issue is to print both the names and the contents of the arguments. The following code displays both the names and the values of each of the arguments (this example also appears with the downloadable source as `GetExceptionArguments2.py`):

```
import sys

try:
   File = open('myfile.txt')
```

(continued)

(continued)

```
except IOError as e:
   for Entry in dir(e):
      if (not Entry.startswith("_")):
         try:
            print(Entry, " = ", e.__getattribute__(Entry))
         except AttributeError:
            print("Attribute ", Entry, " not accessible.")
else:
   print("File opened as expected.")
   File.close();
```

In this case, you begin by getting a listing of the attributes associated with the error argument object using the `dir()` function. The output of the `dir()` function is a list of strings containing the names of the attributes that you can print. Only those arguments that don't start with an underscore (_) contain useful information about the exception. However, even some of those entries are inaccessible, so you must encase the output code in a second `try...except` block (see the "Nested exception handling" section, later in the chapter, for details).

The attribute name is easy because it's contained in `Entry`. To obtain the value associated with that attribute, you must use the `__getattribute__()` function and supply the name of the attribute you want. When you run this code, you see both the name and the value of each of the attributes supplied with a particular error argument object. In this case, the actual output is as follows:

```
args  =  (2, 'No such file or directory')
Attribute  characters_written  not accessible.
errno  =  2
filename  =  myfile.txt
strerror  =  No such file or directory
winerror  =  None
with_traceback  =  <built-in method with_traceback of
   FileNotFoundError object at 0x0000000003416DC8>
```

Handling multiple exceptions with a single except clause

Most applications can generate multiple exceptions for a single line of code. This fact demonstrated earlier in the chapter with the `BasicException1.py` example. How you handle the multiple exceptions depends on your goals for the application, the types of exceptions, and the relative skill of your users. Sometimes when working with a less skilled user, it's simply easier to say that the application experienced a nonrecoverable error and then log the details into a log file in the application directory or a central location.

Using a single `except` clause to handle multiple exceptions works only when a common source of action fulfills the needs of all the exception types. Otherwise, you need to handle each exception individually. The following steps show how to handle multiple exceptions using a single `except` clause. This example also appears with the downloadable source code as `MultipleException1.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   Value = int(input("Type a number between 1 and 10:
         "))
except (ValueError, KeyboardInterrupt):
   print("You must type a number between 1 and 10!")
else:

   if (Value > 0) and (Value <= 10):
      print("You typed: ", Value)
   else:
      print("The value you typed is incorrect!")
```

This code is very much like the `BasicException1.py`. However, notice that the `except` clause now sports both a `ValueError` and a `KeyboardInterrupt` exception. In addition, these exceptions appear within parentheses and are separated by commas.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks you to type a number between 1 and 10.

4. **Type** Hello **and press Enter.**

   The application displays an error message (refer to Figure 9-1).

5. **Perform Steps 3 and 4 again, but type** 22 **instead of Hello.**

   The application outputs the expected range error message (refer to Figure 9-2).

6. **Perform Steps 3 and 4 again, but press Ctrl+C, Cmd+C, or the alternative for your platform instead of typing anything.**

   You see the error message that's usually associated with error input (refer to Figure 9-1).

7. **Perform Steps 3 and 4 again, but type** 7 **instead of Hello.**

   This time, the application finally reports that you've provided a correct value of 7.

Handling multiple exceptions with multiple except clauses

When working with multiple exceptions, it's usually a good idea to place each exception in its own `except` clause. This approach allows you to provide custom handling for each exception and makes it easier for the user to know precisely what went wrong. Of course, this approach is also a lot more work.

The following steps demonstrate how to perform exception handling using multiple `except` clauses. This example also appears with the downloadable source code as `MultipleException2.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   Value = int(input("Type a number between 1 and 10:
        "))
except ValueError:
   print("You must type a number between 1 and 10!")
except KeyboardInterrupt:
   print("You pressed Ctrl+C!")
else:

   if (Value > 0) and (Value <= 10):
      print("You typed: ", Value)
   else:
      print("The value you typed is incorrect!")
```

Notice the use of multiple `except` clauses in this case. Each `except` clause handles a different exception. You can use a combination of techniques, with some `except` clauses handling just one exception and other `except` clauses handling multiple exceptions. Python lets you use the approach that works best for the error-handling situation.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks you to type a number between 1 and 10.

4. **Type** Hello **and press Enter.**

   The application displays an error message (refer to Figure 9-1).

5. **Perform Steps 3 and 4 again, but type** 22 **instead of Hello.**

   The application outputs the expected range error message (refer to Figure 9-2).

6. **Perform Steps 3 and 4 again, but press Ctrl+C, Cmd+C, or the alternative for your platform instead of typing anything.**

   The application outputs a specific message that tells the user what went wrong, as shown in Figure 9-6.

7. **Perform Steps 3 and 4 again, but type** 7 **instead of Hello.**

   This time, the application finally reports that you've provided a correct value of 7.

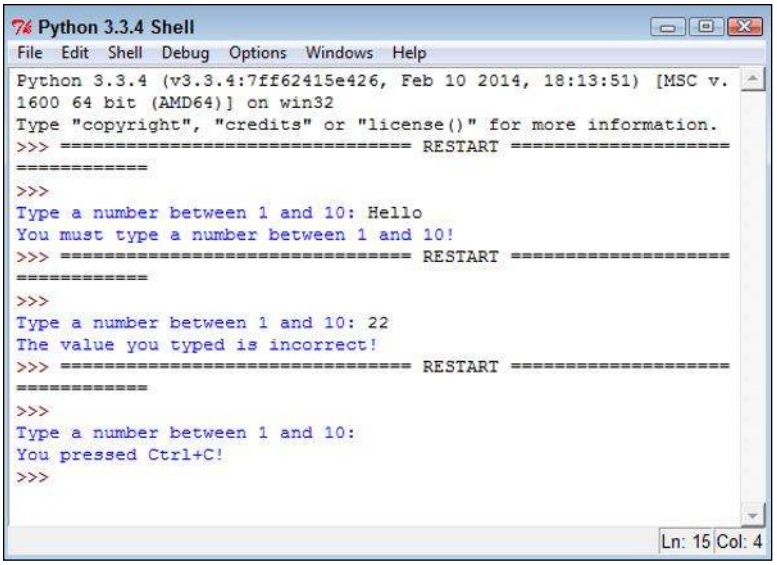

Figure 9-6: Using multiple `except` clauses makes specific error messages possible.

Handling more specific to less specific exceptions

One strategy for handling exceptions is to provide specific `except` clauses for all known exceptions and generic `except` clauses to handle unknown exceptions. You can see the exception hierarchy that Python uses at `https://docs.python.org/3.3/library/exceptions.html#exception-hierarchy`. When viewing this chart, `BaseException` is the uppermost exception. Most exceptions are derived from `Exception`. When working through math errors, you can use the generic `ArithmeticError` or a more specific `ZeroDivisionError` exception.

Python evaluates `except` clauses in the order in which they appear in the source code file. The first clause is examined first, the second clause is examined second, and so on. The following steps help you examine an example that demonstrates the importance of using the correct exception order. In this case, you perform tasks that result in math errors. This example also appears with the downloadable source code as `MultipleException3.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   Value1 = int(input("Type the first number: "))
   Value2 = int(input("Type the second number: "))
   Output = Value1 / Value2
except ValueError:
   print("You must type a whole number!")
except KeyboardInterrupt:
   print("You pressed Ctrl+C!")
except ArithmeticError:
   print("An undefined math error occurred.")
except ZeroDivisionError:
   print("Attempted to divide by zero!")
else:
   print(Output)
```

   The code begins by obtaining two inputs: `Value1` and `Value2`. The first two `except` clauses handle unexpected input. The second two `except` clauses handle math exceptions, such as dividing by zero. If everything goes well with the application, the `else` clause executes, which prints the result of the operation.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks you to type the first number.

4. **Type** Hello **and press Enter.**

   As expected, Python displays the `ValueError` exception message. However, it always pays to check for potential problems.

5. **Choose Run⇨Run Module again.**

   You see a Python Shell window open. The application asks you to type the first number.

6. **Type** 8 **and press Enter.**

   The application asks you to enter the second number.

7. **Type** 0 **and press Enter.**

   You see the error message for the `ArithmeticError` exception, as shown in Figure 9-7. What you should actually see is the `ZeroDivisionError` exception because it's more specific than the `ArithmeticError` exception.

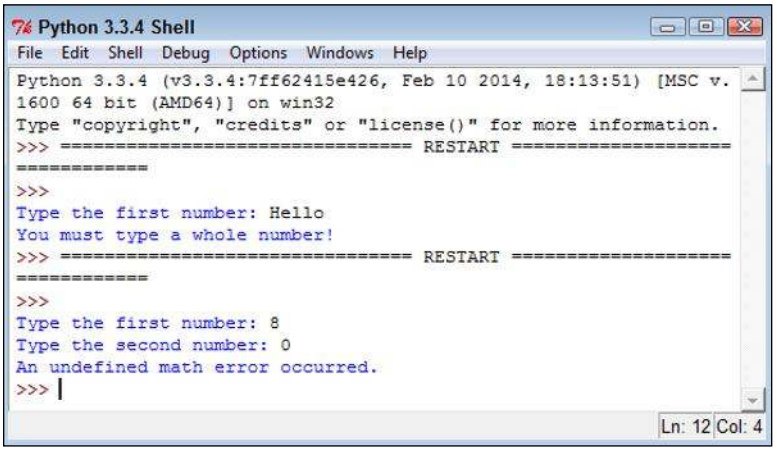

Figure 9-7: The order in which Python processes exceptions is important.

8. **Reverse the order of the two exceptions so that they look like this:**

```
except ZeroDivisionError:
   print("Attempted to divide by zero!")
except ArithmeticError:
   print("An undefined math error occurred.")
```

9. **Perform Steps 5 through 7 again.**

   This time, you see the `ZeroDivisionError` exception message because the exceptions appear in the correct order.

10. **Perform Steps 5 through 7 again, but type** 2 **for the second number instead of 0.**

    This time, the application finally reports an output value of 4.0, as shown in Figure 9-8.

Notice that the output shown in Figure 9-8 is a floating-point value. Division results in a floating-point value unless you specify that you want an integer output by using the floor division operator (//).

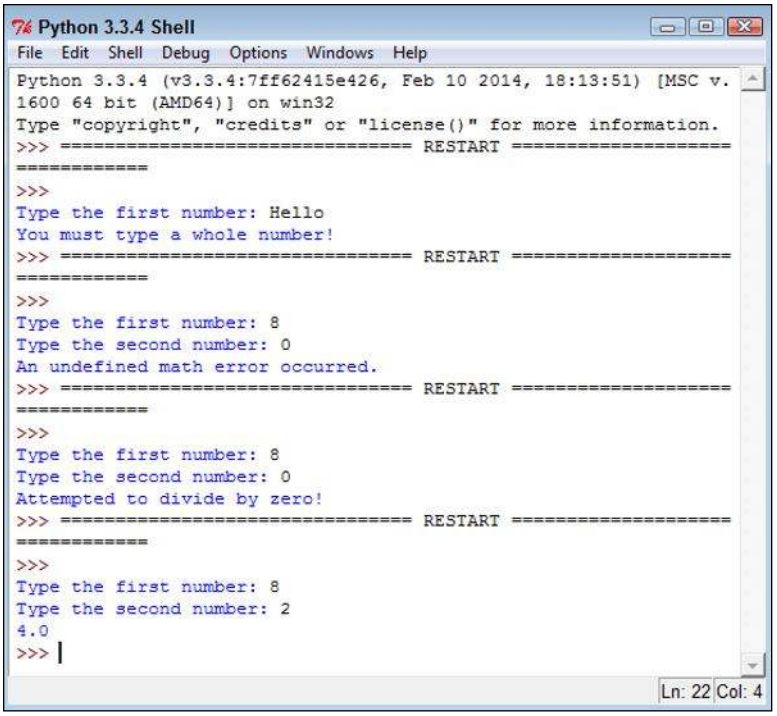

Figure 9-8: Providing usable input results in a usable output.

Nested exception handling

Sometimes you need to place one exception-handling routine within another in a process called *nesting*. When you nest exception-handling routines, Python tries to find an exception handler in the nested level first and then moves to the outer layers. You can nest exception-handling routines as deeply as needed to make your code safe.

One of the more common reasons to use a dual layer of exception-handling code is when you want to obtain input from a user and need to place the input code in a loop to ensure that you actually get the required information. The following steps demonstrate how this sort of code might work. This example also appears with the downloadable source code as `MultipleException4.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
TryAgain = True

while TryAgain:

   try:
      Value = int(input("Type a whole number. "))
   except ValueError:
      print("You must type a whole number!")

      try:
         DoOver = input("Try again (y/n)? ")
      except:
         print("OK, see you next time!")
         TryAgain = False
      else:
         if (str.upper(DoOver) == "N"):
            TryAgain = False

   except KeyboardInterrupt:
      print("You pressed Ctrl+C!")
      print("See you next time!")
      TryAgain = False
   else:
      print(Value)
      TryAgain = False
```

The code begins by creating an input loop. Using loops for this type of purpose is actually quite common in applications because you don't want the application to end every time an input error is made. This is a simplified loop, and normally you create a separate function to hold the code.

When the loop starts, the application asks the user to type a whole number. It can be any integer value. If the user types any non-integer value or presses Ctrl+C, Cmd+C, or another interrupt key combination, the exception-handling code takes over. Otherwise, the application prints the value that the user supplied and sets `TryAgain` to `False`, which causes the loop to end.

A `ValueError` exception can occur when the user makes a mistake. Because you don't know why the user input the wrong value, you have to ask if the user wants to try again. Of course, getting more input from the user could generate another exception. The inner `try . . . except` code block handles this secondary input.

Notice the use of the `str.upper()` function when getting character input from the user. This function makes it possible to receive `y` or `Y` as input and accept them both. Whenever you ask the user for character input, it's a good idea to convert lowercase characters to uppercase so that you can perform a single comparison (reducing the potential for error).

The `KeyboardInterrupt` exception displays two messages and then exits automatically by setting `TryAgain` to `False`. The `KeyboardInterrupt` occurs only when the user presses a specific key combination designed to end the application. The user is unlikely to want to continue using the application at this point.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks the user to input a whole number.

4. **Type** Hello **and press Enter.**

   The application displays an error message and asks whether you want to try again.

5. **Type** Y **and press Enter.**

   The application asks you to input a whole number again, as shown in Figure 9-9.

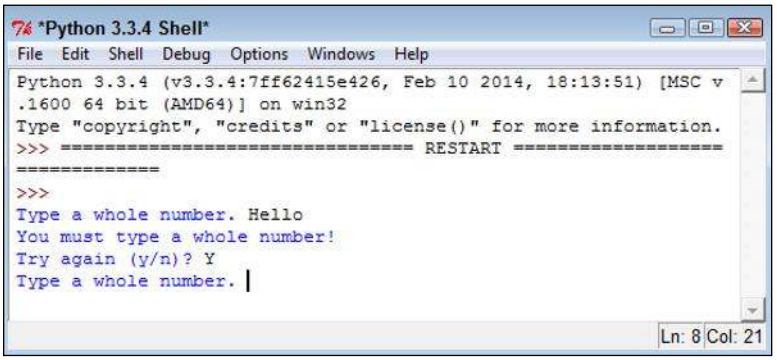

Figure 9-9: Using a loop means that the application can recover from the error.

6. **Type** 5.5 **and press Enter.**

   The application again displays the error message and asks whether you want to try again.

7. **Press Ctrl+C, Cmd+C, or another key combination to interrupt the application.**

   The application ends, as shown in Figure 9-10. Notice that the message is the one from the inner exception. The application never gets to the outer exception because the inner exception handler provides generic exception handling.

8. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application asks the user to input a whole number.

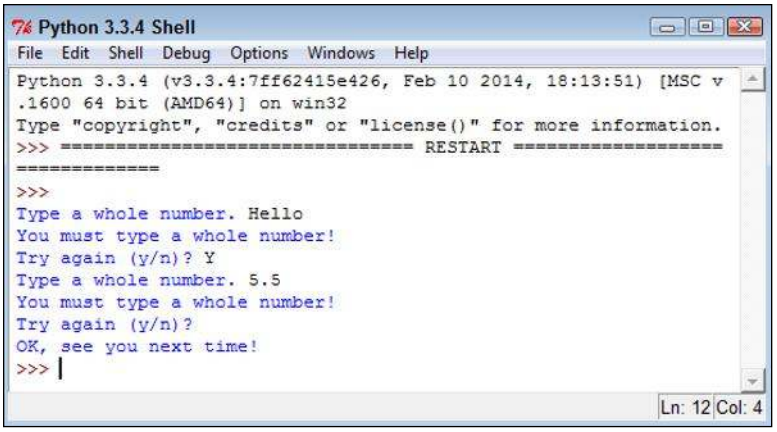

Figure 9-10: The inner exception handler provides secondary input support.

9. **Press Ctrl+C, Cmd+C, or another key combination to interrupt the application.**

   The application ends, as shown in Figure 9-11. Notice that the message is the one from the outer exception. In Steps 7 and 9, the user ends the application by pressing an interrupt key. However, the application uses two different exception handlers to address the problem.

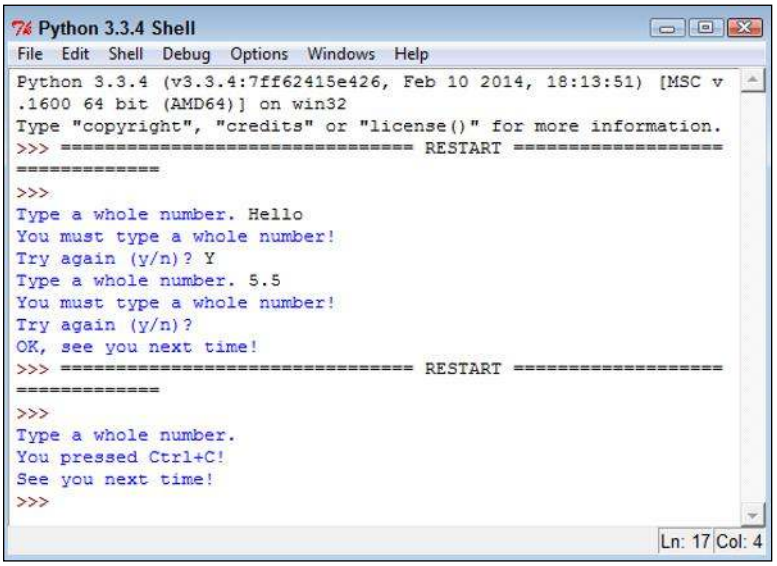

Figure 9-11: The outer exception handler provides primary input support.

Raising Exceptions

So far, the examples in this chapter have reacted to exceptions. Something happens and the application provides error-handling support for that event. However, situations arise for which you may not know how to handle an error event during the application design process. Perhaps you can't even handle the error at a particular level and need to pass it up to some other level to handle. In short, in some situations, your application must generate an exception. This act is called *raising* (or sometimes *throwing*) the exception. The following sections describe common scenarios in which you raise exceptions in specific ways.

Raising exceptions during exceptional conditions

The example in this section demonstrates how you raise a simple exception — that it doesn't require anything special. The following steps simply create the exception and then handle it immediately. This example also appears with the downloadable source code as `RaiseException1.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   raise ValueError
except ValueError:
   print("ValueError Exception!")
```

   You wouldn't ever actually create code that looks like this, but it shows you how raising an exception works at its most basic level. In this case, the `raise` call appears within a `try . . . except` block. A basic raise call simply provides the name of the exception to raise (or throw). You can also provide arguments as part of the output to provide additional information.

   Notice that this `try . . . except` block lacks an `else` clause because there is nothing to do after the call. Although you rarely use a `try . . . except` block in this manner, you can. You may encounter situations like this one sometimes and need to remember that adding the `else` clause is purely optional. On the other hand, you must add at least one `except` clause.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application displays the expected exception text, as shown in Figure 9-12.

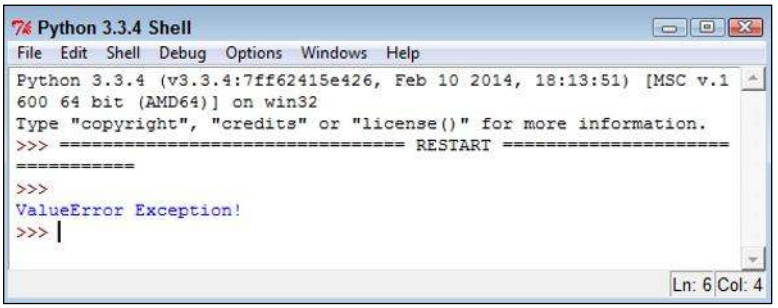

Figure 9-12: Raising an exception only requires a call to `raise`.

Passing error information to the caller

Python provides exceptionally flexible error handling in that you can pass information to the *caller* (the code that is calling your code) no matter which exception you use. Of course, the caller may not know that the information is available, which leads to a lot of discussion on the topic. If you're working with someone else's code and don't know whether additional information is available, you can always use the technique described in the "Obtaining a list of exception arguments" sidebar earlier in this chapter to find it.

You may have wondered whether you could provide better information when working with a `ValueError` exception than with an exception provided natively by Python. The following steps show that you can modify the output so that it does include helpful information. This example also appears with the downloadable source code as `RaiseException2.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
try:
   Ex = ValueError()
   Ex.strerror = "Value must be within 1 and 10."
   raise Ex
except ValueError as e:
   print("ValueError Exception!", e.strerror)
```

The `ValueError` exception normally doesn't provide an attribute named `strerror` (a common name for string error), but you can add it simply by assigning a value to it as shown. When the example raises the exception, the `except` clause handles it as usual but obtains access to the attributes using `e`. You can then access the `e.strerror` member to obtain the added information.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application displays an expanded `ValueError` exception, as shown in Figure 9-13.

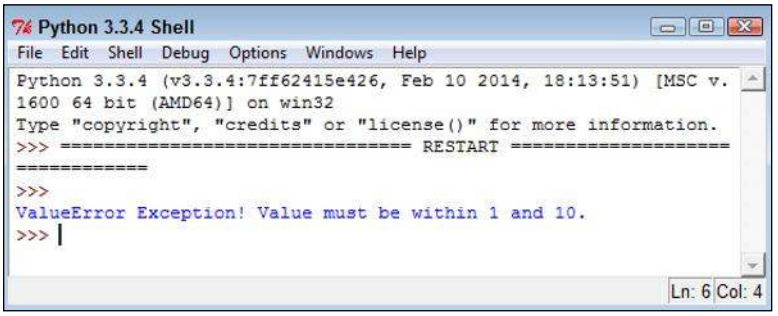

Figure 9-13: It's possible to add error information to any exception.

Creating and Using Custom Exceptions

Python provides a wealth of standard exceptions that you should use whenever possible. These exceptions are incredibly flexible, and you can even modify them as needed (within reason) to meet specific needs. For example, the "Passing error information to the caller" section of this chapter demonstrates how to modify a `ValueError` exception to allow for additional data. However, sometimes you simply must create a custom exception because none of the standard exceptions will work. Perhaps the exception name just doesn't tell the viewer the purpose that the exception serves. You may need a custom exception for specialized database work or when working with a service.

The example in this section is going to seem a little complicated for now because you haven't worked with classes before. Chapter 14 introduces you to classes and helps you understand how they work. If you want to skip this section until after you read Chapter 14, you can do so without any problem.

The example in this section shows a quick method for creating your own exceptions. To perform this task, you must create a class that uses an existing exception as a starting point. To make things a little easier, this example creates an exception that builds upon the functionality provided by the `ValueError` exception. The advantage of using this approach rather than the one shown in the "Passing error information to the caller" section, the preceding section in this chapter, is that this approach tells anyone who follows you precisely what the addition to the `ValueError` exception is; additionally, it makes the modified exception easier to use. This example also appears with the downloadable source code as `CustomException.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
class CustomValueError(ValueError):
   def __init__(self, arg):
      self.strerror = arg
      self.args = {arg}

try:
   raise CustomValueError("Value must be within 1 and
        10.")
except CustomValueError as e:
   print("CustomValueError Exception!", e.strerror)
```

   This example essentially replicates the functionality of the example in the "Passing error information to the caller" section of the chapter. However, it places the same error in both `strerror` and `args` so that the developer has access to either (as would normally happen).

   The code begins by creating the `CustomValueError` class that uses the `ValueError` exception class as a starting point. The `__init__()` function provides the means for creating a new instance of that class. Think of the class as a blueprint and the instance as the building created from the blueprint.

   Notice that the `strerror` attribute has the value assigned directly to it, but `args` receives it as an array. The `args` member normally contains an array of all the exception values, so this is standard procedure, even when `args` contains just one value as it does now.

   The code for using the exception is considerably easier than modifying `ValueError` directly. All you do is call `raise` with the name of the exception and the arguments you want to pass, all on one line.

3. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application displays the letter sequence, along with the letter number, as shown in Figure 9-14.

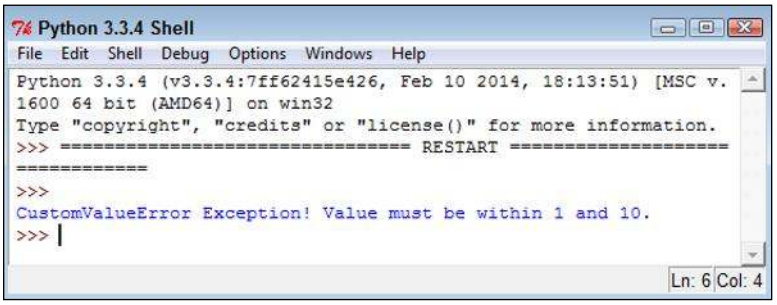

Figure 9-14: Custom exceptions can make your code easier to read.

Using the finally Clause

Normally you want to handle any exception that occurs in a way that doesn't cause the application to crash. However, sometimes you can't do anything to fix the problem, and the application is most definitely going to crash. At this point, your goal is to cause the application to crash gracefully, which means closing files so that the user doesn't lose data and performing other tasks of that nature. Anything you can do to keep damage to data and the system to a minimum is an essential part of handling data for a crashing application.

The `finally` clause is part of the crashing-application strategy. You use this clause to perform any required last-minute tasks. Normally, the `finally` clause is quite short and uses only calls that are likely to succeed without further problem. It's essential to close the files, log the user off, and perform other required tasks, and then let the application crash before something terrible happens (such as a total system failure). With this necessity in mind, the following steps show a simple example of using the `finally` clause. This example also appears with the downloadable source code as `ExceptionWithFinally.py`.

1. **Open a Python File window.**

   You see an editor in which you can type the example code.

2. **Type the following code into the window — pressing Enter after each line:**

```
import sys

try:
   raise ValueError
   print("Raising an exception.")
except ValueError:
   print("ValueError Exception!")
   sys.exit()
finally:
   print("Taking care of last minute details.")

print("This code will never execute.")
```

In this example, the code raises a `ValueError` exception. The `except` clause executes as normal when this happens. The call to `sys.exit()` means that the application exits after the exception is handled. Perhaps the application can't recover in this particular instance, but the application normally ends, which is why the final `print()` function call won't ever execute.

The `finally` clause code always executes. It doesn't matter whether the exception happens or not. The code you place in this block needs to be common code that you always want to execute. For example, when working with a file, you place the code to close the file into this block to ensure that the data isn't damaged by remaining in memory rather than going to disk.

3. **Choose Run⇨Run Module.**

You see a Python Shell window open. The application displays the `except` clause message and the `finally` clause message, as shown in Figure 9-15. The `sys.exit()` call prevents any other code from executing.

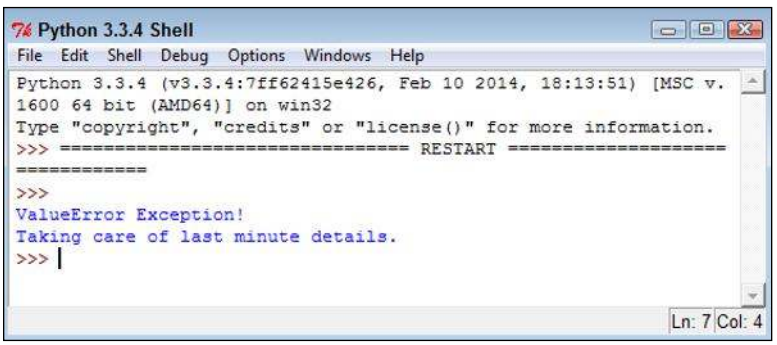

Figure 9-15: Use the `finally` clause to ensure specific actions take place before the application ends.

4. **Comment out the `raise ValueError` call by preceding it with two pound signs, like this:**

   ```
   ##raise ValueError
   ```

   Removing the exception will demonstrate how the `finally` clause actually works.

5. **Save the file to disk to ensure that Python sees the change.**

6. **Choose Run⇨Run Module.**

   You see a Python Shell window open. The application displays a series of messages, including the `finally` clause message, as shown in Figure 9-16. This part of the example shows that the `finally` clause always executes, so you need to use it carefully.

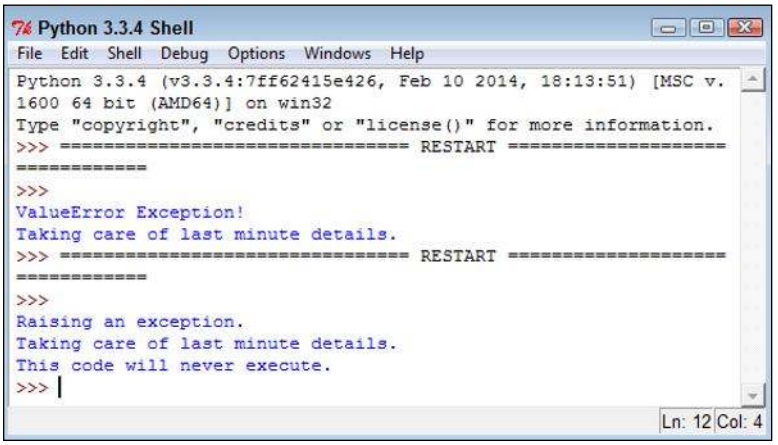

Figure 9-16: It's essential to remember that the `finally` clause always executes.