Exceptions

This lab covers

- [Understanding exceptions]
- [Handling exceptions in Python]
- [Using the [with] keyword]

This lab discusses exceptions, which are language features specifically aimed at handling unusual circumstances during the execution of a program. The most common use for exceptions is to handle errors that arise during the execution of a program, but they can also be used effectively for many other purposes. Python provides a comprehensive set of exceptions, and new ones can be defined by users for their own purposes.

The concept of exceptions as an error-handling mechanism has been around for some time. C and Perl, the most commonly used systems and scripting languages, don't provide any exception capabilities, and even programmers who use languages such as C++, which does include exceptions, are often unfamiliar with them. This lab doesn't assume familiarity with exceptions on your part but instead provides detailed explanations.

Introduction to exceptions

The following sections provide an introduction to exceptions and how they're used. If you're already familiar with exceptions, you can skip directly to "[Exceptions in Python]" ([section 14.2]).

General philosophy of errors and exception handling

Any program may encounter errors during its execution. For the purposes of illustrating exceptions, I look at the case of a word processor that writes files to disk and that therefore may run out of disk space before all of its data is written. There are various ways of coming to grips with this problem.

Solution 1: Don't handle the problem

The simplest way to handle this disk-space problem is to assume that there'll always be adequate disk space for whatever files you write and that you needn't worry about it. Unfortunately, this option seems to be the most commonly used. It's usually tolerable for small programs dealing with small amounts of data, but it's completely unsatisfactory for more mission-critical programs.

Solution 2: All functions return success/failure status

The next level of sophistication in error handling is realizing that errors will occur and defining a methodology using standard language mechanisms for detecting and handling them. There are numerous ways to do this, but a typical method is to have each function or procedure return a status value that indicates whether that function or procedure call executed successfully. Normal results can be passed back in a call-by-reference parameter.

Consider how this solution might work with a hypothetical word-processing program. Assume that the program invokes a single high-level function, [save_to_file], to save the current document to file. This function calls subfunctions to save different parts of the entire document to the file, such as [save_text_to_file] to save the actual document text, [save_prefs_to_file] to save user preferences for that document, [save_formats_to_file] to save user-defined formats for the document, and so forth. Any of these subfunctions may in turn call its own subfunctions, which save smaller pieces to the file. At the bottom are built-in system functions, which write primitive data to the file and report on the success or failure of the file-writing operations.

You could put error-handling code into every function that might get a disk-space error, but that practice makes little sense. The only thing the error handler will be able to do is put up a dialog box telling the user that there's no more disk space and asking the user to remove some files and save again. It wouldn't make sense to duplicate this code everywhere you do a disk write. Instead, put one piece of error-handling code into the main disk-writing function: [save_to_file].

Unfortunately, for [save_to_file] to be able to determine when to call this error-handling code, every function it calls that writes to disk must itself check for disk space errors and return a status value indicating the success or failure of the disk write. In addition, the [save_to_file] function must explicitly check every call to a function that writes to disk, even though it doesn't care about which function fails. The code, using C-like syntax, looks something like this:

```
const ERROR = 1;
const OK = 0;
int save_to_file(filename) {
    int status;
    status = save_prefs_to_file(filename);
    if (status == ERROR) {
        ...handle the error...
    }
    status = save_text_to_file(filename);
    if (status == ERROR) {
        ...handle the error...
    }
    status = save_formats_to_file(filename);
    if (status == ERROR) {
        ...handle the error...
    }
    .
    .
    .
}
int save_text_to_file(filename) {
    int status;
    status = ...lower-level call to write size of text...
    if (status == ERROR) {
        return(ERROR);
    }
    status = ...lower-level call to write actual text data...
    if (status == ERROR) {
        return(ERROR);
    }
    .
    .
    .
}
```

The same applies to [save_prefs_to_file], [save_formats_to_file], and all other functions that either write to [filename] directly or (in any way) call functions that write to [filename].

Under this methodology, code to detect and handle errors can become a significant portion of the entire program, because every function and procedure containing calls that might result in an error needs to contain code to check for an error. Often, programmers don't have the time or the energy to put in this type of complete error checking, and programs end up being unreliable and crash-prone.

Solution 3: The exception mechanism

It's obvious that most of the error-checking code in the previous type of program is largely repetitive: The code checks for errors on each attempted file write and passes an error status message back up to the calling procedure if an error is detected. The disk-space error is handled in only one place: the top-level [save_to_file]. In other words, most of the error-handling code is plumbing code that connects the place where an error is generated with the place

where it's handled. What you really want to do is get rid of this plumbing and write code that looks something like this:

```
def save_to_file(filename)
    try to execute the following block
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    except that, if the disk runs out of space while
        executing the above block, do this
        ...handle the error...

def save_text_to_file(filename)
    ...lower-level call to write size of text...
    ...lower-level call to write actual text data...
    .
    .
    .
```

The error-handling code is completely removed from the lower-level functions; an error (if it occurs) is generated by the built-in file writing routines and propagates directly to the [save_to_file] routine, where your error-handling code will (presumably) take care of it. Although you can't write this code in C, languages that offer exceptions permit exactly this sort of behavior---and of course, Python is one such language. Exceptions let you write clearer code and handle error conditions better.

A more formal definition of exceptions

The act of generating an exception is called *raising* or *throwing* an exception. In the previous example, all exceptions are raised by the disk-writing functions, but exceptions can also be raised by any other functions or can be explicitly raised by your own code. In the previous example, the low-level disk-writing functions (not seen in the code) would throw an exception if the disk were to run out of space.

The act of responding to an exception is called *catching* an exception, and the code that handles an exception is called *exception-handling code* or just an *exception handler*. In the example, the [except that...] line catches the disk-write exception, and the code that would be in place of the [...handle the error...] line would be an exception handler for disk-write (out of space) exceptions. There may be other exception handlers for other types of exceptions or even other exception handlers for the same type of exception but at another place in your code.

Handling different types of exceptions

Depending on exactly what event causes an exception, a program may need to take different actions. An exception raised when disk space is exhausted needs to be handled quite differently from an exception that's raised if you run out of memory, and both of these exceptions are completely different from an exception that arises when a divide-by-zero error occurs. One way to handle these different types of exceptions is to globally record an error message indicating the cause of the exception, and have all exception handlers examine this error message and take appropriate action. In practice, a different method has proved to be much more flexible.

Rather than defining a single kind of exception, Python, like most modern languages that implement exceptions, defines different types of exceptions corresponding to various problems that may occur. Depending on the underlying event, different types of exceptions may be raised. In addition, the code that catches exceptions may be told to catch only certain types. This feature is used in the pseudocode in solution 3 earlier in this lab that said [except that, if the disk runs out of space . . ., do this]; this pseudocode specifies that this particular exception-

handling code is interested only in disk-space exceptions. Another type of exception wouldn't be caught by that exception-handling code. That exception would be caught by an exception handler that was looking for numeric exceptions, or (if no such exception handler existed) it would cause the program to exit prematurely with an error.

Exceptions in Python

The remaining sections of this lab talk specifically about the exception mechanisms built into Python. The entire Python exception mechanism is built around an object-oriented paradigm, which makes it both flexible and expandable. If you aren't familiar with object-oriented programming (OOP), you don't need to learn object-oriented techniques to use exceptions.

An exception is an object generated automatically by Python functions with a [raise] statement. After the object is generated, the [raise] statement, which raises an exception, causes execution of the Python program to proceed in a manner different from what would normally occur. Instead of proceeding with the next statement after the [raise] or whatever generated the exception, the current call chain is searched for a handler that can handle the generated exception. If such a handler is found, it's invoked and may access the exception object for more information. If no suitable exception handler is found, the program aborts with an error message.

Easier to ask forgiveness than permission

The way that Python thinks about handling error situations in general is different from that common in languages such as Java, for example. Those languages rely on checking for possible errors as much as possible before they occur, since handling exceptions after they occur tends to be costly in various ways. This style is described in the first section of this lab and is sometimes described as a look before you leap (LBYL) approach.

Python, on the other hand, is more likely to rely on exceptions to deal with errors after they occur. Although this reliance may seem to be risky, if exceptions are used well, the code is less cumbersome and easier to read, and errors are dealt with only as they occur. This Pythonic approach to handling errors is often described by the phrase "easier to ask forgiveness than permission" (EAFP).

Types of Python exceptions

It's possible to generate different types of exceptions to reflect the actual cause of the error or exceptional circumstance being reported. Python 3.6 provides several exception types:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError
            OverflowError
           ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
            ModuleNoteFoundError
        LookupError
            IndexError
            KeyError
        MemoryError
```

```
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError
                ConnectionAbortedError
                ConnectionRefusedError
                ConnectionResetError
            FileExistsError
            FileNotFoundError
            InterruptedError
            IsADirectoryError
            NotADirectoryError
            PermissionError
            ProcessLookupError
            TimeoutError
        ReferenceError
        RuntimeError
            NotImplementedError
            RecursionError
        SyntaxError
            IndentationError
                TabError
        SystemError
        TypeError
        ValueError
            UnicodeError
                UnicodeDecodeError
                UnicodeEncodeError
                UnicodeTranslateError
        Warning
            DeprecationWarning
            PendingDeprecationWarning
            RuntimeWarning
            SyntaxWarning
            UserWarning
            FutureWarning
            ImportWarning
            UnicodeWarning
            BytesWarningException
            ResourceWarning
```

The Python exception set is hierarchically structured, as reflected by the indentation in this list of exceptions. As you saw in a previous chapter, you can obtain an alphabetized list from the [__builtins__] module.

Each type of exception is a Python class, which inherits from its parent exception type. But if you're not into OOP yet, don't worry about that. An [IndexError], for example, is also a [LookupError] and (by inheritance) an [Exception] and also a [BaseException].

This hierarchy is deliberate: Most exceptions inherit from [Exception], and it's strongly recommended that any user-defined exceptions also subclass [Exception], not [BaseException]. The reason is that if you have code set up like this

```
try:
    # do stuff
except Exception:
    # handle exceptions
```

you could still interrupt the code in the [try] block with Ctrl-C without triggering the exception-handling code, because the [KeyboardInterrupt] exception is *not* a subclass of [Exception].

You can find an explanation of the meaning of each type of exception in the documentation, but you'll rapidly become acquainted with the most common types as you program!

Raising exceptions

Exceptions are raised by many of the Python built-in functions:

```
>>> alist = [1, 2, 3]
>>> element = alist[7]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Error-checking code built into Python detects that the second input line requests an element at a list index that doesn't exist and raises an [IndexError] exception. This exception propagates all the way back to the top level (the interactive Python interpreter), which handles it by printing out a message stating that the exception has occurred.

Exceptions may also be raised explicitly in your own code through the use of the [raise] statement. The most basic form of this statement is

```
raise exception(args)
```

The [exception(args)] part of the code creates an exception. The arguments to the new exception are typically values that aid you in determining what happened---something that I discuss next. After the exception has been created, [raise] throws it upward along the stack of Python functions that were invoked in getting to the line containing the [raise] statement. The new exception is thrown up to the nearest (on the stack) exception catcher looking for that type of exception. If no catcher is found on the way to the top level of the program, the program terminates with an error or (in an interactive session) causes an error message to be printed to the console.

Try the following:

```
>>> raise IndexError("Just kidding")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: Just kidding
```

The use of [raise] here generates what at first glance looks similar to all the Python list-index error messages you've seen so far. Closer inspection reveals this isn't the case. The actual error reported isn't as serious as those other ones.

The use of a string argument when creating exceptions is common. Most of the built-in Python exceptions, if given a first argument, assume that the argument is a message to be shown to you as an explanation of what happened. This isn't always the case, though, because each exception type is its own class, and the arguments expected when a new exception of that class is created are determined entirely by the class definition. Also, programmer-defined exceptions, created by you or by other programmers, are often used for reasons other than error handling; as such, they may not take a text message.

Catching and handling exceptions

The important thing about exceptions isn't that they cause a program to halt with an error message. Achieving that function in a program is never much of a problem. What's special about exceptions is that they don't have to cause the program to halt. By defining appropriate exception handlers, you can ensure that commonly encountered exceptional circumstances don't cause the program to fail; perhaps they display an error message to the user or do something else, perhaps even fix the problem, but they don't crash the program.

The basic Python syntax for exception catching and handling is as follows, using the [try], [except], and sometimes [else] keywords:

```
try:
    body
except exception_type1 as var1:
    exception_code1
except exception_type2 as var2:
    exception_code2
    .
    .
    .
except:
    default_exception_code
else:
    else_body
finally:
    finally_body
```

A [try] statement is executed by first executing the code in the [body] part of the statement. If this execution is successful (that is, no exceptions are thrown to be caught by the [try] statement), the [else_body] is executed, and the [try] statement is finished. Because there is a [finally] statement, [finally_body] is executed. If an exception is thrown to the [try], the [except] clauses are searched sequentially for one whose associated exception type matches that which was thrown. If a matching [except] clause is found, the thrown exception is assigned to the variable named after the associated exception type, and the exception code body associated with the matching exception is executed. If the line [except exception_type] as [var]: matches some thrown exception [exc], the variable [var] is created, and [exc] is assigned as the value of [var] before the exception-handling code of the [except] statement is executed. You don't need to put in [var]; you can say something like [except exception_type:], which still catches exceptions of the given type but doesn't assign them to any variable.

If no matching [except] clause is found, the thrown exception can't be handled by that [try] statement, and the exception is thrown farther up the call chain in hope that some enclosing [try] will be able to handle it.

The last [except] clause of a [try] statement can optionally refer to no exception types at all, in which case it handles all types of exceptions. This technique can be convenient for some debugging and extremely rapid prototyping but generally isn't a good idea: all errors are hidden by the [except] clause, which can lead to some confusing behavior on the part of your program.

The [else] clause of a [try] statement is optional and rarely used. This clause is executed if and only if the [body] of the [try] statement executes without throwing any errors.

The [finally] clause of a [try] statement is also optional and executes after the [try], [except], and [else] sections have executed. If an exception is raised in the [try] block and isn't handled by any of the [except] blocks, that exception is raised again after the [finally] block executes. Because the [finally] block always executes, it gives you a chance to include code to clean up after any exception handling by closing files, resetting variables, and so on.

Try this: Catching exceptions

Write code that gets two numbers from the user and divides the first number by the second. Check for and catch the exception that occurs if the second number is zero ([ZeroDivisionError]).

Defining new exceptions

You can easily define your own exception. The following two lines do this for you:

```
class MyError(Exception):
    pass
```

This code creates a class that inherits everything from the base [Exception] class. But you don't have to worry about that if you don't want to.

You can raise, catch, and handle this exception like any other exception. If you give it a single argument (and you don't catch and handle it), it's printed at the end of the traceback:

```
>>> raise MyError("Some information about what went wrong")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: Some information about what went wrong
```

This argument, of course, is available to a handler you write as well:

```
try:
    raise MyError("Some information about what went wrong")
except MyError as error:
    print("Situation:", error)
```

The result is

```
Situation: Some information about what went wrong
```

If you raise your exception with multiple arguments, these arguments are delivered to your handler as a tuple, which you can access through the [args] variable of the error:

```
try:
    raise MyError("Some information", "my_filename", 3)
except MyError as error:
    print("Situation: {0} with file {1}\n error code: {2}".format(
        error.args[0],
 error.args[1], error.args[2]))
```

The result is

```
Situation: Some information with file my_filename
error code: 3
```

Because an exception type is a regular class in Python and happens to inherit from the root [Exception] class, it's a simple matter to create your own subhierarchy of exception types for use by your own code. You don't have to worry about this process on a first read of this book. You can always come back to it after you've read [chapter 15]. Exactly how you create your own exceptions depends on your particular needs. If you're writing a small program that may generate only a few unique errors or exceptions, subclass the main [Exception] class as you've done here. If you're writing a large, multifile code library with a special goal in mind---say, weather forecasting---you may decide to define a unique class called [WeatherLibraryException] and then define all the unique exceptions of the library as subclasses of [WeatherLibraryException].

Quick Check: Exceptions as classes

If [MyError] inherits from [Exception], what is the difference between [except Exception as e] and [except MyError as e]?

Debugging programs with the assert statement

The [assert] statement is a specialized form of the [raise] statement:

```
assert expression, argument
```

The [AssertionError] exception with the optional [argument] is raised if the [expression] evaluates to [False] and the system variable [__debug__] is [True]. The [__debug__] variable defaults to [True] and is turned off by starting the Python interpreter with the [-O] or [-OO] option or by setting the system variable [PYTHONOPTIMIZE] to [True]. The optional argument can be used to include an explanation of the assertion.

The code generator creates no code for assertion statements if [__debug__] is [False]. You can use [assert] statements to instrument your code with debug statements during development and leave them in the code for possible future use with no runtime cost during regular use:

```
>>> x = (1, 2, 3)
>>> assert len(x) > 5, "len(x) not > 5"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: len(x) not > 5
```

Try this: The assert statement

Write a simple program that gets a number from the user and then uses the [assert] statement to raise an exception if the number is zero. Test to make sure that the [assert] statement fires; then turn it off, using one of the methods mentioned in this section.

The exception inheritance hierarchy

In this section, I expand on an earlier notion that Python exceptions are hierarchically structured and on what that structure means in terms of how [except] clauses catch exceptions.

The following code

```
try:
    body
except LookupError as error:
    exception code
except IndexError as error:
    exception code
```

catches two types of exceptions: [IndexError] and [LookupError]. It just so happens that [IndexError] is a subclass of [LookupError]. If [body] throws an [IndexError], that error is first examined by the [except LookupError as error:] line, and because an [IndexError] is a [LookupError] by inheritance, the first [except] succeeds. The second [except] clause is never used because it's subsumed by the first [except] clause.

Conversely, flipping the order of the two [except] clauses could potentially be useful; then the first clause would handle [IndexError] exceptions, and the second clause would handle any [LookupError] exceptions that aren't [IndexError] errors.

Example: a disk-writing program in Python

In this section, I revisit the example of a word-processing program that needs to check for disk out-of-space conditions as it writes a document to disk:

```
def save_to_file(filename) :
    try:
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    except IOError:
        ...handle the error...
def save_text_to_file(filename):
    ...lower-level call to write size of text...
    ...lower-level call to write actual text data...
    .
    .
    .
```

Notice how unobtrusive the error-handling code is; it's wrapped around the main sequence of disk-writing calls in the [save_to_file] function. None of the subsidiary disk-writing functions needs any error-handling code. It would be easy to develop the program first and add error-handling code later. That's often what programmers do, although this practice isn't the optimal ordering of events.

As another note of interest, this code doesn't respond specifically to disk-full errors; rather, it responds to [IOError] exceptions, which Python's built-in functions raise automatically whenever they can't complete an I/O request, for whatever reason. That's probably satisfactory for your needs, but if you need to identify disk-full conditions, you can do a couple of things. The [except] body can check to see how much room is available on disk. If the disk is out of space, clearly, the problem is a disk-full problem and should be handled in this [except] body; otherwise, the code in the [except] body can throw the [IOError] farther up the call chain to be handled by some other [except]. If that solution isn't sufficient, you can do something more extreme, such as going into the C source for the Python disk-writing functions and raising your own [DiskFull] exceptions as necessary. I don't recommend the latter option, but it's nice to know that this possibility exists if you need to use it.

Example: exceptions in normal evaluation

Exceptions are most often used in error handling but can also be remarkably useful in certain situations involving what you'd think of as normal evaluation. Consider the problems in implementing something that works like a spreadsheet. Like most spreadsheets, it would have to permit arithmetic operations involving cells, and it would also permit cells to contain values other than numbers. In such an application, blank cells used in a numerical calculation might be considered to contain the value [0], and cells containing any other nonnumeric string might be considered invalid and represented as the Python value [None]. Any calculation involving an invalid value should return an invalid value.

The first step is to write a function that evaluates a string from a cell of the spreadsheet and returns an appropriate value:

```
def cell_value(string):
    try:
        return float(string)
    except ValueError:
        if string == "":
            return 0
```

```
        else:
            return None
```

Python's exception-handling ability makes this function a simple one to write. The code tries to convert the string from the cell to a number and return it in a [try] block using the [float] built-in function. [float] raises the [ValueError] exception if it can't convert its string argument to a number, so the code catches that exception and returns either [0] or [None], depending on whether the argument string is empty or nonempty.

The next step is handling the fact that some of the arithmetic might have to deal with a value of [None]. In a language without exceptions, the normal way to do this is to define a custom set of arithmetic functions, which check their arguments for [None], and then use those functions rather than the built-in arithmetic functions to perform all of the spreadsheet arithmetic. This process is time-consuming and error-prone, however, and it leads to slow execution because you're effectively building an interpreter in your spreadsheet. This project takes a different approach. All the spreadsheet formulas can actually be Python functions that take as arguments the x and y coordinates of the cell being evaluated and the spreadsheet itself, and calculate the result for the given cell by using standard Python arithmetic operators, using [cell_value] to extract the necessary values from the spreadsheet. You can define a function called [safe_apply] that applies one of these formulas to the appropriate arguments in a [try] block and returns either the formula's result or [None], depending on whether the formula evaluated successfully:

```
def safe_apply(function, x, y, spreadsheet):
    try:
        return function(x, y, spreadsheet)
    except TypeError:
        return None
```

These two changes are enough to integrate the idea of an empty ([None]) value into the semantics of the spreadsheet. Trying to develop this ability without the use of exceptions is a highly educational exercise.

Where to use exceptions

Exceptions are natural choices for handling almost any error condition. It's an unfortunate fact that error handling is often added when the rest of the program is largely complete, but exceptions are particularly good at intelligibly managing this sort of after-the-fact error-handling code (or, more optimistically, when you're adding more error handling after the fact).

Exceptions are also highly useful in circumstances where a large amount of processing may need to be discarded after it becomes obvious that a computational branch in your program has become untenable. The spreadsheet example is one such case; others are branch-and-bound algorithms and parsing algorithms.

Quick Check: Exceptions

Do Python exceptions force a program to halt?

Suppose that you want accessing a dictionary [x] to always return [None] if a key doesn't exist in the dictionary (that is, if a [KeyError] exception is raised). What code would you use?

Context managers using the with keyword

Some situations, such as reading files, follow a predictable pattern with a set beginning and end. In the case of reading from a file, quite often the file needs to be open only one time: while data is being read. Then the file can be closed. In terms of exceptions, you can code this kind of file access like this:

```
try:
    infile = open(filename)
    data = infile.read()
```

```
finally:
    infile.close()
```

Python 3 offers a more generic way of handling situations like this: context managers. *Context managers* wrap a block and manage requirements on *entry* and *departure* from the block and are marked by the [with] keyword. File objects are context managers, and you can use that capability to read files:

```
with open(filename) as infile:
    data = infile.read()
```

These two lines of code are equivalent to the five previous lines. In both cases, you know that the file will be closed immediately after the last read, whether or not the operation was successful. In the second case, closure of the file is also assured because it's part of the file object's context management, so you don't need to write the code. In other words, by using [with] combined with a context management (in this case a file object), you don't need to worry about the routine cleanup.

As you might expect, it's also possible to create your own context managers if you need them. You can learn a bit more about how to create context managers and the various ways they can be manipulated by checking out the documentation for the [contextlib] module of the standard library.

Context managers are great for things like locking and unlocking resources, closing files, committing database transactions, and so on. Since their introduction, context managers have become standard best practice for such use cases.

Quick Check: Context managers

Assume that you're using a context manager in a script that reads and/or writes several files. Which of the following approaches do you think would be best?

- [Put the entire script in a block managed by a [with] statement.]
- [Use one [with] statement for all file reads and another for all file writes.]
- [Use a [with] statement each time you read a file or write a file (for each line, for example).]
- [Use a [with] statement for each file that you read or write.]

Lab 14: Custom exceptions

Think about the module you wrote in [chapter 9] to count word frequencies. What errors might reasonably occur in those functions? Refactor those functions to handle those exception conditions appropriately.

Summary

- [Python's exception-handling mechanism and exception classes provide a rich system to handle runtime errors in your code.]
- [By using [try], [except], [else], and [finally] blocks, and by selecting and even creating the types of exceptions caught, you can have very fine-grained control over how exceptions are handled and ignored.]
- [Python's philosophy is that errors shouldn't pass silently unless they're explicitly silenced.]
- [Python exception types are organized in a hierarchy because exceptions, like all objects in Python, are based on classes.]