Chapter 34. Exception Coding Details

The prior chapter provided a quick look at exception-related statements in action. Here, we're going to dig a bit deeper—this chapter provides fuller coverage of exception-processing syntax in Python. Specifically, we'll explore the details behind the `try`, `raise`, `assert`, and `with` statements. Although these statements are mostly straightforward, you'll find that they offer powerful tools for dealing with exceptional conditions in Python code.

The try Statement

First up, the `try` statement is how your code catches exceptions. In short, if an exception occurs while running this statement's main block, the program jumps back to run one of the statement's *handlers* and continues from there. Its handlers may be specified by `except`, `else`, `finally`, and `except*` *clauses* nested in the `try`, and separate rules apply to these clauses' syntax, and their valid combinations.

This is a simple model on the surface, but the `try` statement's handler clauses have disjoint purposes, and its rules for valid combinations mean that it comes in distinct flavors. Because of this, we'll approach this subject by exploring the `try`'s common roles in isolation first and putting their pieces together later as a combined statement. This parallels the fact that `try` really *was* separate statements in Python's dim past, but our focus here is on its unified present.

Although technically part of the `try`, we'll also defer the `except*` clause until the next chapter, partly because it encroaches on that chapter's exception-*object* topic, but mostly because this is a tool that complicates the `try` story substantially for an extension that's rarely useful in practice. Our priority here is learning the fundamentals.

try Statement Clauses

When you write a `try` statement, a variety of clauses can appear after and below the `try` header. Table 34-1 summarizes all the possible forms as both reference and preview. We've already seen that `except` clauses catch exceptions and `finally` clauses run on the way out. New here, `else` clauses run if no exceptions are encountered, and `except*` clauses process exception groups and support all `except` forms except the empty.

Formally, a `try` must use at least one of the clauses in Table 34-1. There may be any number of `except` clauses, but you can code `else` only if there is at least one `except`, and there can be only one `else` and one `finally`. A `finally` can appear in the same statement as `except` and `else`, with ordering rules given later in this chapter.

Table 34-1. `try` statement clauses and forms

Clause form	Interpretation
`except:`	Catch all (or all other) exception types
`except` *`name`*`:`	Catch a specific exception only
`except` *`name`* `as` *`var`*`:`	Catch the listed exception and assign its instance
`except (`*`name1`*`,` *`name2`*`):`	Catch any of the exceptions listed in a tuple
`except (`*`name1`*`,` *`name2`*`) as` *`var`*`:`	Catch any listed exception and assign its instance
`else:`	Run if no exceptions are raised in the `try` block
`finally:`	Always perform this block on exit, exception or not
`except*` … *`nonempty except forms`* …`:`	Catch multiple exceptions in a group (Chapter 35)

We'll explore the `as` *`var`* part available in some of Table 34-1's clauses in more detail when we meet the `raise` statement later in this chapter because it provides access to the object raised as an exception via `var`. Before all

that, let's get started by examining the more common clauses of Table 34-1 more closely.

The except and else Clauses

Syntactically, the `try` is a compound, multipart statement. It starts with a `try` header line, followed by a block of (usually) indented statements, which is followed by clauses that each identify a condition to be handled and give a block of statements to handle it. In its most common form, `try` is coded with one or more `except` clauses and an optional `else` clause at the end. You associate the words `try`, `except`, and `else` by indenting them to the same level (i.e., lining them up vertically), like this:

```
try:
    statements              # Run this main action firs
except name1:
    statements              # Run if name1 is raised du
except (name2, name3):
    statements              # Run if name2 or name3 occ
except name4 as var:
    statements              # Run if name4 is raised, c
except:
    statements              # Run for all other excepti
else:
    statements              # Run if no exception was r
```

Semantically, the block under the `try` header in this statement represents the *main action* of the statement—the code you're trying to run, and wrapping in exception handlers. The rest of the statement defines the handlers themselves: `except` clauses give handlers for exceptions raised during the `try` block, and the optional `else` clause gives a handler run if *no* exceptions occur in the `try` block.

Within a `try`, each `except` names exceptions to catch: a *single* exception catches just that exception, a tuple catches *any* exception in the tuple, and an `except` that omits the exception altogether matches *all* (or all *other*) exceptions. Each nonempty `except` can also give a variable name after `as` to be assigned the exception object raised by Python or `raise` statements; again, we'll explore this option ahead.

How try statements work

Operationally, here's how `try` statements are run. When a `try` statement is entered, Python records the current program context so it can return to it if an exception occurs. The statements nested under the `try` header are run first. What happens next depends on whether an exception is raised while the `try` block's statements are running, and whether a raised exception matches any of those that the `try` is watching for:

Exception and match

If an exception occurs while the `try` block's statements are running, and the exception *matches* one that the statement names, Python jumps back to the `try` and runs the statements under its topmost `except` clause that matches the raised exception, after assigning the raised exception object to the variable named by `as` in the clause (if present). After the `except` block runs, control resumes below the entire `try` statement. If the `except` block itself raises another exception, the propagation process is started anew from this point in the code.

Exception and no match

If an exception occurs while the `try` block's statements are running, but the exception *does not* match one that the statement names, the exception is propagated up to the next most recently entered `try` statement that matches the exception; if no such matching `try` statement can be found and the search reaches the top level of the program, Python prints a default error message and terminates the program (unless it's the REPL).

No exception

If an exception does *not* occur while the `try` block's statements are running, Python runs the statements under the `else` clause (if present), and control then resumes below the entire `try` statement. If the `else` block itself raises another exception, it kicks off the propagation process again.

In sum, `except` clauses catch any matching exceptions that happen while the `try` block is running, and the `else` clause runs only if no exceptions happen while the `try` block runs. Exceptions raised are *matched* to exceptions named in `except` clauses by *class* relationships we'll explore both ahead and in the next chapter (brief: a subclass matches its superclass),

and the *empty* `except` clause with no exception name matches any exception.

In effect, `except` clauses are *focused* exception handlers—they catch exceptions that occur only within the statements in the associated `try` block. However, as the `try` block's statements can call functions coded elsewhere in a program, the source of an exception may very well be outside the code of the `try` statement itself.

In fact, a `try` block might invoke arbitrarily large amounts of program code—including code that may have `try` statements of its own, which will be searched first when exceptions occur. In other words, because `try` statements can nest at runtime, where an exception goes depends on the code run before it, a phenomenon we'll explore in Chapter 36.

If a `finally` clause is added to a `try` , its code block is run for all three of the cases listed previously, as you'll see ahead. First, though, let's take a look at some common variations of exception-catching clauses.

Catching many exceptions with a tuple

Per the fourth and fifth entries in Table 34-1, `except` clauses that list *many* exceptions in a parenthesized tuple catch *any* of the listed exceptions. Because Python looks for a match within a given `try` by inspecting the `except` clauses from *top to bottom*, the tuple version has the same effect as listing each exception in its own `except` clause, but you have to code the common statement body associated with each only once.

Here's a partial example of multiple `except` clauses at work, which demos just how specific your handlers can be:

```
try:
    …
except NameError:
    …
except IndexError:
    …
except (AttributeError, TypeError, SyntaxError):
    …
```

If an exception is raised while this `try` block is running, Python returns to the `try` and searches for the first `except` that names the exception raised. It inspects clauses from top to bottom—and left to right along the way—and runs the statements under the first clause that matches. If none match, the exception is propagated past this `try` .

Note that *parentheses* are required around the tuple in the "any" form, and using an `as` in this form lets you check which exception occurred when you listed many:

```
try:
    …
except (AttributeError, TypeError, SyntaxError) as What
    …and check What…
```

To learn more about both `as` , as well as what happens when no `except` matches, we must move on.

Catching all exceptions with empties and Exception

Per the first entry in Table 34-1, `except` clauses that list *no* exception name catch *all* exceptions not previously listed in the `try` statement. That is, if you want to code a general "catchall" handler to be run when no other `except` clause matches the exception raised, an empty `except` does the trick:

```
try:
    …
except NameError:
    …                   # Handle NameError
except IndexError:
    …                   # Handle IndexError
except:
    …                   # Handle all other exceptions
else:
    …                   # Handle the no-exception case
```

The empty `except` clause is a sort of *wildcard* feature—because it catches everything, adding it to the mix allows your handlers to be as general or specific as you like. In some scenarios, this form may be more convenient

than listing all possible exceptions in a `try` —especially when working interactively in a REPL, or writing code that must recover no matter what occurs. For example, the following catches *everything* by not listing anything:

```
try:
    …
except:
    …                       # Catch all possible exceptions
```

That being shown, the empty `except` can also cause problems. It may catch *unexpected* exceptions, and intercept events unrelated to your code and *required* by another handler. For example, even *system exit* calls and Ctrl+C key-combination interrupts in Python work by triggering exceptions, and you usually want these to pass.

Perhaps worse, the empty `except` may also catch genuine *programming mistakes* for which you probably want to see an error message. Otherwise, you may not even know that a bug exists until it's too late to avoid a user's report. We'll revisit this as a gotcha at the end of this part of the book. For now, the standard "use with care" applies.

Python provides an alternative that solves at least one of these problems—catching the built-in `Exception` has almost the same effect as an empty `except` , but won't catch exceptions related to system exits and Ctrl+C:

```
try:
    …
except Exception:
    …                       # Catch all possible exceptions
```

We'll explore how this form does its magic formally in the next chapter when we study exception classes. In short, it works because exceptions match if they are a *subclass* of one named in an `except` clause, and `Exception` is a superclass of all the exceptions you should generally catch this way. This form has most of the same convenience of the empty `except` without the risk of catching exit events and also allows you to check the exception raised via `as` . While better, though, it also has some of the same risks—it may still mask and silently ignore programming errors.

The opening snippet of this section deliberately listed an `else` clause, to call out that it is *not* a catchall like the empty `except` —an understandable source of confusion for `try` newcomers. The next section dissects the difference.

Catching the no-exception case with else

All told, `else` can be used in *three* places in Python: in `if` selections, `for` and `while` loops, and `try` exception handlers. In the latter, `else` is run when *no* exception occurs—not for *unmatched* exceptions (that's what the prior section's empty `except` is for). This `else` role may seem different than in `if` and loops, but this context differs.

The need for an `else` clause in `try` is not always obvious to Python beginners. Without it, though, there is no direct way to tell whether the flow of control has proceeded past a `try` statement because no exception was raised, or because an exception occurred and was handled. Either way, we wind up after the `try` :

```
try:
    …run code…
except IndexError:
    …handle exception…
# Did we get here because the try failed or not?
```

Of course, we could initialize, set, and check a Boolean flag to know what happened, which adds lines of admin code. Much like the way `else` clauses in loops make the exit cause more apparent (for exits sans `break`), the `else` clause provides syntax in `try` that makes the outcome unambiguous with minimal extra code:

```
try:
    …run code…
except IndexError:
    …handle exception…
else:
    …no exception occurred…
```

You can *almost* emulate an `else` clause by moving its code into the `try` block:

```
try:
    …run code…
    …no exception occurred…
except IndexError:
    …handle exception…
```

This can lead to incorrect exception classifications, though. If the "no exception occurred" action itself causes an `IndexError`, it will register as a failure of the `try` block and erroneously trigger the exception handler below the `try` (unlikely perhaps, but true). By using an explicit `else` clause instead, you make the logic more obvious and guarantee that `except` handlers will run only for real failures in the code you're wrapping in a `try`, not for failures in the `else` no-exception case's action.

Example: Default behavior

Because the control flow through a program may be easier to capture in Python than in English, let's run some simple examples that further illustrate exception basics with real code in files.

As noted, exceptions not caught by `try` statements percolate up to the top level of the Python process and run Python's default exception-handling logic (i.e., Python terminates the running program and prints a standard error message). To illustrate, module file *crashware.py* coded in Example 34-1 generates a divide-by-zero exception—by design.

Example 34-1. crashware.py

```
def gobad(x, y):
    return x / y

def gosouth(x):
    print(gobad(x, 0))

if __name__ == '__main__': gosouth(1)
```

Because the program ignores the exception it triggers, Python kills the program and prints a message (edited here to condense paths and drop some interfaces' code-pointer lines for space):

```
$ python3 crashware.py
Traceback (most recent call last):
  File "/…/LP6E/Chapter34/crashware.py", line 7, in <mc
    if __name__ == '__main__': gosouth(1)
  File "/…/LP6E/Chapter34/crashware.py", line 5, in gos
    print(gobad(x, 0))
  File "/…/LP6E/Chapter34/crashware.py", line 2, in gob
    return x / y
ZeroDivisionError: division by zero
```

This message consists of a stack trace ("Traceback") and the name of and details about the exception that was raised. The stack trace lists all lines active when the exception occurred, from oldest to newest. Note that because this code was written in a file instead of a REPL, the file and line-number information is more useful here. For example, we can see that the bad divide happens at the last entry in the trace—line 2 of the file *crashware.py*, a `return` statement.

Because Python detects and reports all errors at runtime by raising exceptions like this, exceptions are intimately bound up with the ideas of *error handling* and *debugging* in general. If you've worked through this book's examples, you've undoubtedly seen an exception or two along the way—even typos usually generate a `SyntaxError` or other exception when a file is imported or executed (that's when the code compiler is run).

By default, programming errors generate a useful error display like the one just shown, which helps you track down the problem. In some interfaces, this message today even comes with pointers to offending expressions, as well as speculative but mandatory "Did you?" tips (whose merit you may wish to reweigh later in your Python career):

```
$ python3
>>> import crashware
>>> crashware.gobad(1, 0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/…/LP6E/Chapter34/crashware.py", line 2, in gob
    return x / y
           ~~^~~
ZeroDivisionError: division by zero
```

Often, this standard error message is all you need to resolve problems in your code. For more heavy-duty debugging jobs, you can catch exceptions with `try` statements, or use debugging tools introduced in Chapter 3, such as the `pdb` standard-library module. See "Debugging Python Code" for more related tips.

Example: Catching built-in exceptions

Python's error checking and default exception handling is often exactly what you want: especially for code in a top-level script file, an error often *should* terminate your program immediately. For many programs, there is no need to be more specific about errors in your code.

Sometimes, though, you'll want to catch errors and recover from them instead. If you don't want your program terminated when Python raises an exception, simply catch it by wrapping the program logic in a `try` . This is an important capability for programs such as network servers, which must keep running persistently. For example, the code in Example 34-2, file *kaboom.py*, catches and recovers from the `TypeError` Python raises immediately when you try to concatenate a list and a string (remember, the `+` operator expects the same sequence type on both sides).

Example 34-2. kaboom.py

```
def kaboom(x, y):
    print(x + y)                          # Trigger TypeE

def serve(n=2):                           # Simulate long
    for i in range(n):
        try:
            kaboom([1, 2], 'hack')
        except TypeError:                 # Catch and rec
            print('Hello world!')
        print('Resuming here...')         # Continue here

if __name__ == '__main__': serve()
```

When the exception occurs in the function `kaboom` , control jumps to the `try` statement's `except` clause, which prints a message. Since an exception is "dead" after it's been caught like this, the program continues

executing below the `try` rather than being terminated by Python. In effect, the code processes and clears the error, and your script recovers:

```
$ python3 kaboom.py
Hello world!
Resuming here...
Hello world!
Resuming here...
```

Keep in mind that once you've caught an error, control resumes at the place where you caught it (i.e., after the `try`); there is no direct way to go back to the place where the exception occurred (here, in the function `kaboom`). This makes exceptions more like simple *jumps* than function calls—there is no way to "return" to the scene of the crime.

The finally Clause

The other main flavor of the `try` statement is for coding *finalization* (a.k.a. termination) actions and is really related to exceptions only incidentally. If a `finally` clause is included in a `try`, Python will always run its block of statements "on the way out" of the `try` statement—whether an exception occurred while the `try` block was running or not. That is, this clause doesn't *catch* exceptions, it works around them.

When used in isolation, this flavor's general form is this:

```
try:
    statements          # Run this action first
finally:
    statements          # Always run this code on the
```

When a `finally` appears in `try`, Python begins by running the statement block associated with the `try` header line as usual. What happens next depends on whether an exception occurs during the `try` block, and what other clauses are present:

Exception and match

If an exception occurs during the `try` block's run and is matched by an `except` clause, Python first runs the matching `except` block

and then runs the `finally` block. After both finish, the program then resumes below the entire `try` statement. The `finally` is also run if the `except` raises a new exception.

Exception and no match

If an exception occurs during the `try` block's run but is *not* caught by an `except` , Python still comes back and runs the `finally` block, but it then propagates the exception up to a previously entered `try` or the top-level default handler. That is, `finally` is run even if an exception is raised and uncaught, but unlike an `except` , the `finally` does not terminate the exception—it continues being raised after the `finally` block runs.

No exception

If an exception does *not* occur while the `try` block is running, Python first runs the `else` block (if present) and then runs the `finally` block. After both finish, the program then resumes below the entire `try` statement. The `finally` is also run if the `else` raises a new exception.

The `try` / `finally` form is useful when you want to be completely sure that an action will happen after some code runs, regardless of the exception behavior of the code. In practice, it allows you to specify cleanup actions that always must occur, such as file closes and server disconnects where required.

Technically speaking, `finally` can appear in the same statement as `except` and `else` , so there is really a single `try` statement with many optional clauses. Because of its distinct role, though, as well as its ordering rules we will meet in a moment, the `finally` clause may be best thought of as a distinct tool. Whether mixed or not, `finally` serves the same purpose —to specify cleanup actions that must always be run, regardless of any exceptions.

As you'll also see later in this chapter, the `with` statement and its context managers provide an object-based way to do similar work for exit actions. Unlike `finally` , this statement also supports entry actions, but it is limited in scope to objects that implement the context-manager protocol it employs.

Example: Coding termination actions with try/finally

We coded some simple `try` / `finally` examples in the prior chapter. Example 34-3 lists a more tangible example that illustrates a typical role for

this statement.

Example 34-3. closer.py

```
class MyError(Exception): pass

def stuff(file):
    file.write('Hello?')             # May be delayed i
    raise MyError()                  # <= Enable or dis

if __name__ == '__main__':
    file = open('temp.txt', 'w')     # Open an output f
    try:
        stuff(file)                  # Raises exceptior
    finally:
        file.close()                 # Always close fil
    print('Am I reached?')           # Continue here or
```

When the function in this code raises its exception, the control flow jumps back and runs the `finally` block to close the file. The exception is then propagated on to either another `try` or the default top-level handler, which prints the standard error message and shuts down the program. Hence, the statement after this `try` is never reached:

```
$ python3 closer.py
Traceback (most recent call last):
  File "/…/LP6E/Chapter34/closer.py", line 10, in <modu
    stuff(file)                      # Raises exceptior
  File "/…/LP6E/Chapter34/closer.py", line 5, in stuff
    raise MyError()                  # <= Enable or dis
MyError
```

If the function here did *not* raise an exception (e.g., by disabling its `raise` line with an added `#`), the program would still execute the `finally` block to close the file, but it would then continue below the entire `try` statement:

```
$ python3 closer.py
Am I reached?
```

In this specific case, we've wrapped a call to a file-processing function in a `try` with a `finally` clause to make sure that the file is always closed, and thus finalized, whether the function triggers an exception or not. This way, later code can be sure that the file's output buffer's content has been flushed from memory to disk. A similar code structure can guarantee that server connections are closed, GUI windows are closed, and so on.

As we learned in Chapter 9, file objects are automatically closed on garbage collection in standard Python (CPython); this is especially useful for temporary files that we don't assign to variables. However, it's not always easy to predict when garbage collection will occur, especially in larger programs or alternative Python implementations with differing garbage collection policies. The `try` statement makes file closes more explicit and predictable: it ensures that the file will be closed on block exit, regardless of whether an exception occurs or not.

This particular example's function isn't all that useful (it always raises an exception!), but wrapping calls in `try` / `finally` statements is a good way to ensure that your closing-time termination activities always run. All bets are off if Python itself crashes completely, of course, but this is exceedingly rare; because it detects errors as a program runs, hard crashes are usually caused by linked-in C extension code, outside of Python's scope.

As a preview, notice how the user-defined exception in Example 34-3 is defined with a *class*; as you'll learn more formally in the next chapter, exceptions must all be class instances, for reasonably good causes.

Combined try Clauses

For the first 15 years of Python's tenure (more or less), the `try` statement came in two flavors and was two separate statements—we could either use a `finally` to ensure that cleanup code was always run, or write `except` blocks to catch and recover from specific exceptions and optionally specify an `else` clause for when no exceptions occurred.

That is, the `finally` clause could not be *mixed* with `except` and `else`. This was partly because of implementation issues, and partly because the meaning of mixing the two seemed obscure—catching and recovering from exceptions seemed a disjoint concept from performing cleanup actions.

For better or worse, the two statements eventually merged. Today, we can mix `finally`, `except`, and `else` clauses in the same statement—in part because of similar utility in the Java language (alas, many a programming-language mod owes to imitation). That is, the `try` statement in its most complete form looks like this:

```
try:                        # Combined try statement
    main-action
except Exception1:          # Catch specific exceptions
    handler1
except Exception2:
    handler2
except:                     # Catch all (other) exceptic
    handler3
else:                       # No-exception handler
    handler4
finally:                    # The finally encloses all
    finally-block
```

The code in this statement's *`main-action`* block is executed first, as usual. If that code raises an exception, all the `except` blocks are tested, one after another, for a match to the exception raised: *`handler1`* is run for *`Exception1`*, *`handler2`* for *`Exception2`*, and *`handler3`* for all others. If no exception is raised, *`handler4`* is run.

No matter what's happened previously, the *`finally-block`* is executed once, after the main action block is exited and any handler block has been run. In fact, the code in the *`finally-block`* will be run even if an error or `raise` in an `except` or `else` block causes a new exception to be raised.

As outlined earlier, even in mixed usage like this, the `finally` clause does not end the exception—if an exception is active when the *`finally-block`* is executed, it continues to be propagated after the *`finally-block`* runs, and control jumps somewhere else in the program (to an earlier `try`, or to the default top-level handler). If no exception is active when the `finally` is run, control resumes after the entire `try` statement.

The net effect is that the `finally` is always run, regardless of whether:

- An exception occurred in the main action and was handled.

- An exception occurred in the main action and was not handled.
- No exceptions occurred in the main action.
- A new exception was triggered in one of the handlers.

Again, the `finally` serves to specify cleanup actions that must always occur on the way out of the `try`, regardless of what exceptions have been raised or handled.

Combined-clause syntax rules

When combined like this, the `try` statement must have either an `except` or a `finally`, and the order of its parts must be like this (where "->" means "is followed by"):

```
try -> except -> else -> finally
```

In this, the `else` and `finally` are optional, and there may be zero or more `except`s, but there must be at least one `except` if an `else` appears. Really, the `try` statement consists of two parts: `except`s with an optional `else`, and/or the `finally`.

In fact, it's more accurate to describe the combined `try` statement's syntactic by the following two alternative formats (where square brackets mean optional and star means any number of what precedes it):

```
try:                              # Format 1
    statements
except [type [as value]]:
    statements
[except [type [as value]]:
    statements]*
[else:
    statements]
[finally:
    statements]

try:                              # Format 2
    statements
finally:
    statements
```

Because of these rules, the `else` can appear only if there is at least one `except`, and it's always possible to mix `except` and `finally`, regardless of whether an `else` appears or not. It's also possible to mix `finally` and `else`, but only if an `except` appears too (though the `except` can omit an exception name to catch everything and run a `raise` statement, described later, to reraise the current exception). If you violate any of these (arguably intricate!) ordering rules, Python will raise a syntax error exception before your code runs.

Combining finally and except by nesting

It may help to realize that it's also possible to combine `finally` and `except` clauses in a `try` by syntactically nesting a `try`/`except` in the `try` block of a `try`/`finally` statement. We'll explore this technique more fully in Chapter 36, but the following has the same effect as the combined form shown at the start of this section:

```
try:                            # Nested equivalent to com
    try:
        main-action
    except Exception1:
        handler1
    except Exception2:
        handler2
    except:
        handler3
    else:
        handler4
finally:
    finally-block
```

Again, the `finally` block is always run on the way out, regardless of what happened in the main action and regardless of any exception handlers run in the nested `try` (trace through the four cases listed previously to see how this works the same). Since an `else` always requires an `except`, this nested form even sports the same mixing constraints of the combined form outlined in the preceding section.

However, this nested equivalent seems more obscure to some people and requires more code than the new merged form—though just one four-

character line plus extra indentation. Mixing `finally` into the same statement might make your code easier to write and read, though this also might depend on who you ask.

Combined-clauses example

To demo the effect of mixing `finally` with other `try` clauses, the script listed in Example 34-4, *trycombos.py*, codes four common scenarios, with `print` statements that describe the meaning of each.

Example 34-4. trycombos.py

```
sep = '-' * 45 + '\n'


print(sep + 'EXCEPTION RAISED AND CAUGHT')
try:
    x = 'hack'[99]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')


print(sep + 'EXCEPTION NOT RAISED')
try:
    x = 'hack'[3]
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')


print(sep + 'EXCEPTION NOT RAISED, WITH ELSE')
try:
    x = 'hack'[3]
except IndexError:
    print('except run')
else:
    print('else run')
finally:
    print('finally run')
```

```
print('after run')


print(sep + 'EXCEPTION RAISED BUT NOT CAUGHT')
try:
    x = 1 / 0
except IndexError:
    print('except run')
finally:
    print('finally run')
print('after run')
```

When this code is run, the following output is produced. Trace through the code to see how exception handling produces the output of each of the four tests here:

```
$ python3 trycombos.py
-----------------------------------------------
EXCEPTION RAISED AND CAUGHT
except run
finally run
after run
-----------------------------------------------
EXCEPTION NOT RAISED
finally run
after run
-----------------------------------------------
EXCEPTION NOT RAISED, WITH ELSE
else run
finally run
after run
-----------------------------------------------
EXCEPTION RAISED BUT NOT CAUGHT
finally run
Traceback (most recent call last):
  File "/…/LP6E/Chapter34/trycombos.py", line 38, in <m
    x = 1 / 0
ZeroDivisionError: division by zero
```

This example uses built-in operations in the main action to trigger exceptions (or not), and it relies on the fact that Python always checks for errors as code is running. The next section shows how to raise exceptions manually instead.

The raise Statement

To trigger exceptions explicitly, code `raise` statements. Their general form is simple—a `raise` statement consists of the word `raise`, optionally followed by the class to be raised, or an instance of it:

```
raise instance            # Raise an instance of a class
raise class               # Make and raise an instance o
raise                     # Reraise the most recent exce
```

As mentioned earlier, exceptions are always instances of *classes* today. Hence, the first `raise` form here is the most common—we provide an *instance* directly, either created before the `raise` or within the `raise` statement itself. If we pass a *class* instead, Python calls the class with *no* constructor arguments, to create an instance to be raised; this form is equivalent to adding parentheses after the class reference. The last form reraises the most recently raised exception; it's commonly used in exception handlers to propagate exceptions that have been caught.

NOTE

Blast from the past: Long ago and far away (well, before Python 2.6), exceptions could be identified as simple *string* objects, with an optional associated data item in `raise`. This was replaced with *classes* to support added functionality and categories, as you'll see in the next chapter. Still, strings were a simpler model for simpler roles, didn't require newcomers to learn classes and OOP before exceptions, and didn't force some Python books to put exceptions on hold until Part VII!

Raising Exceptions

To make `raise` more concrete, let's turn to some examples. With built-in exceptions, the following two forms are equivalent—both raise an instance of the exception class named, but the first creates the instance implicitly:

```
raise IndexError              # Class (instance created)
raise IndexError()            # Instance (created in sta
```

We can also create the instance ahead of time—because the `raise` statement accepts any kind of object reference, the following two examples raise `IndexError` just like the prior two:

```
exc = IndexError()            # Create instance ahead of
raise exc

excs = [IndexError, TypeError]
raise excs[0]
```

In fact, the instance provided to `raise` can be had in the `try` that catches it too, per the next section.

The except as hook

When an exception is raised, Python sends the raised instance along with the exception. If a `try` includes an `except` with an `as` clause per Table 34-1, the variable it gives will be assigned the instance raised by `raise` or Python:

```
try:
    …
except IndexError as X:       # X assigned the raised ir
    …
```

The `as` is optional in a `try` handler (if it's omitted, the instance is simply not assigned to a name), but including it allows the handler to access both data in the instance and methods in the exception class.

This model works the same for user-defined exceptions we code with classes—the following, for example, passes to the exception class constructor arguments that become available in the handler through the assigned instance:

```
class MyExc(Exception): pass

try:
    raise MyExc('oops')       # Exception class with cor
except MyExc as X:            # Instance attributes avai
    print(X.args)             # Prints ('oops',)
```

Because this encroaches on the next chapter's topic, though, we'll defer further details until then.

Regardless of their source, exceptions are always identified by class *instance* objects, and at most one is active at any given time (sans the `except*` groups of Chapter 35). Once caught by an `except` clause anywhere in the program, an exception ends and won't propagate to another `try`, unless it's reraised by another `raise` statement or error.

Scopes and except as

We'll study exception objects in more detail in the next chapter. Now that we've seen the `as` variable in action, though, we can finally clarify the related scope issue summarized back in Chapter 17. As mentioned in that chapter, the variable used to access an exception in the `as` clause of an `except` is localized to the `except` block—the variable is not available after the block exits, much like a temporary loop variable in comprehension expressions:

```
$ python3
>>> try:
...     1 / 0
... except Exception as X:              # The "as" locali
...     print(X)
...
division by zero
>>> X
NameError: name 'X' is not defined
```

Unlike comprehension loop variables, though, this variable is *removed* after the `except` block exits. This is done because the variable would otherwise retain a reference to the runtime call stack, which would defer garbage collection and thus retain excess memory space. This removal occurs, though, *even* if you're using the name for other purposes in the surrounding scope, and is a much more extreme policy than that used for comprehensions:

```
>>> X = 99
>>> {X for X in 'hack'}                 # Comprehensions
{'a', 'c', 'k', 'h'}
>>> X
```

```
99

>>> X = 99
>>> try:
...     1 / 0
... except Exception as X:            # But "as" locali
...     print(X)
...
division by zero
>>> X                                 # Where did my X
NameError: name 'X' is not defined
```

Because of this, you should generally use unique variable names in your `try` statement's `except` clauses, even if they are localized by scope. If you do need to reference the exception instance after the `try` statement, simply assign it to another name that won't be automatically removed:

```
>>> try:
...     1 / 0
... except Exception as X:            # Python removes
...     print(X)
...     saveit = X                    # Assign exc to r
...
division by zero
>>> X
NameError: name 'X' is not defined
>>> saveit
ZeroDivisionError('division by zero',)
```

Propagating Exceptions with raise

The `raise` statement is a bit more feature-rich than we've seen thus far. For example, a `raise` that does not list an exception to raise simply reraises the currently active exception. This form is typically used if you need to catch and handle an exception but don't want the exception to die in your handler:

```
>>> try:
...     raise IndexError('code')         # Exceptions r
... except IndexError:
...     print('propagating')
...     raise                            # Reraise most
...
```

```
propagating
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: code
```

Running a `raise` this way reraises the exception and propagates it to a higher handler (or the default handler at the top, which stops the program with a standard error message). Notice how the argument we passed to the exception class shows up in the error messages; you'll learn why this happens in the next chapter.

Exception Chaining: raise from

Exceptions can sometimes be triggered in response to other exceptions—both deliberately and by new program errors. To support full disclosure in such cases, Python also allows `raise` statements to have an optional `from` clause:

```
raise newexception from otherexception
```

When the `from` is used in an *explicit* `raise` request, the expression following `from` specifies another exception class or instance to attach to the `__cause__` attribute of the new exception being raised. If the raised exception is not caught, Python prints both exceptions as part of the standard error message:

```
>>> try:
...     1 / 0
... except Exception as E:
...     raise TypeError('Bad') from E             # Exp
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the followi

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
TypeError: Bad
```

When an exception is raised *implicitly* by a program error inside an exception handler, a similar procedure is followed automatically: the previous exception is attached to the new exception's `__context__` attribute and is again displayed in the standard error message if the exception goes uncaught:

```
>>> try:
...     1 / 0
... except:
...     badname                                    # Imp
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another excepti

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'badname' is not defined
```

In both cases, because the original exception objects thus attached to new exception objects may *themselves* have attached causes, the causality chain can be *arbitrarily long*, and is displayed in full in error messages. That is, error messages might give more than two exceptions. The net effect in both explicit and implicit chaining contexts is to allow programmers to know all exceptions involved when one exception triggers another:

```
>>> try:
...     try:
...         raise IndexError()
...     except Exception as E:
...         raise TypeError() from E
... except Exception as E:
...     raise SyntaxError() from E
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
IndexError

The above exception was the direct cause of the followi

Traceback (most recent call last):
```

```
  File "<stdin>", line 5, in <module>
TypeError

The above exception was the direct cause of the followi

Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
SyntaxError: None
```

Code like the following similarly displays three exceptions, though implicitly triggered by handler errors (its separator lines are "During handling of the above exception…" instead of "The above exception was the direct cause…"):

```
try:
    try:
        1 / 0
    except:
        badname
except:
    open('nonesuch')
```

Exception chains impact error displays, but do not affect the way that exceptions are named and caught in `try` statements: chains are simply recorded in exception-object attributes which may be inspected as usual where useful.

Like the combined `try`, chained exceptions are similar to utility in other languages (including Java and C#) though it's not clear which languages were borrowers. In Python, it's not unusual to see exception chains in error messages, but it is uncommon to create them explicitly with `raise`, so we'll defer to Python's manuals for more details.

As a footnote on this topic, though, Python also provides a way to *stop* exceptions from chaining: a `raise from None` allows the display of the chained exception context to be disabled when needed. This makes for less cluttered error messages in applications that convert between exception types while processing exception chains.

The assert Statement

As a somewhat special case for debugging purposes, Python also includes the `assert` statement in its exceptions toolset. It is mostly just syntactic shorthand for a common `raise` usage pattern, and an `assert` can be thought of as a *conditional* `raise` statement. A statement of the form:

```
assert test, data                # The data part is optic
```

works like the following code:

```
if __debug__:
    if not test:
        raise AssertionError(data)
```

In other words, if the *`test`* evaluates to false, Python raises an exception: the *`data`* item (if it's provided) is used as the exception's constructor argument. Like all exceptions, the built-in `AssertionError` exception will kill your program if it's not caught with a `try`, and the *`data`* item shows up as part of the standard error message:

```
>>> language = 'Java'
>>> assert language.startswith('Py'), "You're using the
AssertionError: You're using the wrong language!
```

As an added feature, `assert` statements are removed from a compiled program's bytecode—and hence not run—if the `-O` Python command-line flag is used to optimize the program. The `__debug__` flag is a built-in and unchangeable name that is automatically set to `True` unless the `-O` flag is used. When `__debug__` is `False` for `-O`, any code predicated on it being `True` is removed, including asserts.

Hence, to disable (and omit) asserts, run code with a command line like `python -O *file*.py`, or generate optimized bytecode before program runs with similar options in the `compileall` standard-library module or `compile` built-in function. See Python's manuals for details on the module and function.

Example: Trapping Constraints (but Not Errors!)

Here's a less politically charged example of `assert` in action. Assertions are typically used to verify program conditions during development. When displayed, their error message text automatically includes source code line information and the value listed in the `assert` statement. Consider the file *asserter.py* in Example 34-5.

Example 34-5. asserter.py

```
def f(x):
    assert x < 0, 'x must be negative'
    return x ** 2
```

Running this normally triggers the assertion error for positive numbers, but running with `-O` does not:

```
$ python3
>>> import asserter
>>> asserter.f(-3)
9
>>> asserter.f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/…/LP6E/Chapter34/asserter.py", line 2, in f
    assert x < 0, 'x must be negative'
AssertionError: x must be negative

$ python3 -O
>>> import asserter
>>> asserter.f(3)
9
```

It's important to keep in mind that `assert` is mostly intended for trapping user-defined constraints, not for catching genuine programming *errors*. Because Python traps programming errors itself, there is usually no need to code `assert` to catch things like out-of-bounds indexes, type mismatches, and zero divides:

```
def reciprocal(x):
    assert x != 0                # A generally useless as
    return 1 / x                 # Python checks for zerc
```

Such `assert` use cases are usually superfluous—because Python raises exceptions on errors automatically, you might as well let it do the job for you. As a rule, you normally don't need to do error checking explicitly in your own code.

Of course, there are exceptions to most rules. As suggested earlier in the book, if a function has to perform long-running or unrecoverable actions before it reaches the place where an exception will be triggered, you still might want to test for errors. Even in this case, though, be careful not to make your tests overly specific or restrictive, or you will limit your code's utility.

For another example of common `assert` usage, see the abstract superclass example in Chapter 29; there, we used `assert` to make calls to undefined methods fail with a message. It's a rare but useful tool.

The with Statement and Context Managers

In addition to the tools we've seen so far, Python includes another that delegates exception-related tasks to objects. The `with` statement is designed to work with *context manager* objects that support a method-based protocol. The combination is similar in spirit to the way that iteration tools like `for` work with methods of the iteration protocol.

The `with` statement is also similar to a "using" statement in the C# language. Although a somewhat optional and advanced tools-oriented topic (and once a candidate for the next part of this book), context managers are lightweight and useful enough to group with the rest of the exception toolset here.

In short, the `with` statement is designed to be an alternative to a common `try`/`finally` usage idiom: like that statement, `with` is in large part intended for specifying termination-time or "cleanup" activities that must run

regardless of whether an exception occurs during the execution of a block of code.

Unlike `try` / `finally` , the `with` statement is based upon a method-call protocol for specifying actions to be run around a block of code. This makes `with` less general, qualifies it as redundant in termination roles, and requires coding classes for objects that do not support its protocol. On the other hand, `with` also handles entry actions, can reduce code size where supported, and allows code contexts to be managed with full OOP.

Python enhances some built-in tools with context managers, such as files that automatically close themselves, thread locks that automatically lock and unlock, and async-function tools that automatically await results per Chapter 20, but programmers can code context managers of their own with classes, too. Let's take a brief look at the statement and its implicit protocol.

Basic with Usage

The basic format of the `with` statement looks like this, with an optional part in square brackets here:

```
with expression [as variable]:
    with-block
```

The statements of the nested *`with-block`* are the main action to be run here. The *`expression`* is assumed to return an object that supports the context-management protocol (more on this protocol in a moment). This object may also return a value that will be assigned to the name *`variable`* if the optional `as` clause is present.

Note that the *`variable`* is not necessarily assigned the result of the *`expression`* ; the result of the *`expression`* is the object that supports the context protocol, and the *`variable`* may be assigned something else intended to be used inside the *`with-block`* . The object returned by the *`expression`* may then run startup code before the block is started, as well as termination code after the block is done—whether the block raised an exception or not.

As noted, some built-in Python objects have been augmented to support the context-management protocol, and so can be used with the `with` statement.

For example, file objects (covered in Chapter 9) have a context manager that automatically closes the file after the `with` block regardless of whether an exception is raised, and regardless of if or when the version of Python running the code may close automatically. In abstract code:

```
with open('somefile.txt') as myfile:
    for line in myfile:
        print(line)
```

Here, the call to `open` returns a simple file object that is assigned to the name `myfile`. We can use `myfile` with the usual file tools—in this case, the file iterator reads line by line in the `for` loop.

However, the `open` result also supports the context-management protocol used by the `with` statement. After this `with` statement has run, the context management machinery guarantees that the file object referenced by `myfile` is automatically closed, even if the `for` loop raised an exception while processing the file.

Although file objects may be automatically closed on garbage collection, it's not always straightforward to know when that will occur, especially when using alternative Python implementations. The `with` statement in this role is an alternative that allows us to be sure that the close will occur automatically after execution of a specific block of code.

As covered earlier, we can achieve a similar effect with the more general and explicit `try`/`finally` idiom, but it requires three more lines of administrative code in this case (four instead of just one):

```
myfile = open('somefile.txt')
try:
    for line in myfile:
        print(line)
finally:
    myfile.close()
```

Of course, we could skip *both* statements, but our file may not be closed if an exception is raised during the `for` loop, and this can matter in long-running programs (we'll revisit such trade-offs in Chapter 36).

As another example, we won't cover Python's multithreading modules in this book, but the lock and condition synchronization objects they define may also be used with the `with` statement because they support the context-management protocol—in this case adding both entry and exit actions around a block. After importing `threading`:

```
lock = threading.Lock()
with lock:
    …access shared resources…
```

Here, the context management machinery guarantees that the lock is automatically *acquired* before the block is executed and *released* once the block is complete, regardless of exception outcomes.

Finally, the `decimal` module introduced in Chapter 5 also uses context managers to simplify saving and restoring the current decimal context, which specifies the precision and rounding characteristics for calculations:

```
with decimal.localcontext() as ctx:
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00
```

After this statement runs, the current thread's context manager state is automatically restored to what it was before the statement began. To do the same with a `try`/`finally`, we would need to save the context before and restore it manually after the nested block.

The Context-Management Protocol

Although some built-in types come with context managers, we can also write new ones of our own. To implement context managers, classes use special methods that fall into the operator-overloading category to tap into the `with` statement. The interface expected of objects used in `with` statements is somewhat complex, and most programmers only need to know how to use existing context managers. For tool builders who might want to write new application-specific context managers, though, let's take a quick look at what's involved.

Here's how the `with` statement actually works:

1. The expression is evaluated, resulting in an object known as a *context manager* that must have `__enter__` and `__exit__` methods.
2. The context manager's `__enter__` method is called. The value it returns is assigned to the variable in the `as` clause if present, or simply discarded otherwise.
3. The code in the nested `with` block is executed.
4. If the `with` block raises an exception, the context manager's `__exit__(type, value, traceback)` method is called with the exception details. These are the same three values returned by `sys.exc_info`, described in the Python manuals and later in this part of the book. If this method returns a false value, the exception is reraised; otherwise, the exception is terminated. The exception should normally be reraised so that it is propagated outside the `with` statement after `__exit__` returns.
5. If the `with` block does *not* raise an exception, the `__exit__` method is still called, but its `type`, `value`, and `traceback` arguments are all passed in as `None`, and its return value is ignored.

Let's look at a quick demo of the protocol in action. The file *withas.py* in Example 34-6 defines a context-manager object that simply traces the entry and exit of the `with` block in any `with` statement it is used for.

Example 34-6. withas.py

```
"A context manager that traces entry and exit of any wi

class TraceBlock:
    def message(self, arg):
        print('running ' + arg)

    def __enter__(self):
        print('[starting with block]')
        return self

    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is None:
            print('[exited normally]\n')
        else:
            print(f'[propagating exception: {exc_type}]
            return False

if __name__ == '__main__':
```

```
    with TraceBlock() as action:
        action.message('test 1')
        print('reached')

    with TraceBlock() as action:
        action.message('test 2')
        raise TypeError
        print('not reached')
```

Notice that this class's `__exit__` method returns `False` to propagate the exception; deleting the `return` statement would have the same effect, as the default `None` return value of functions is false by definition, but explicit is generally better in coding. Also notice that the `__enter__` method returns `self` as the object to assign to the `as` variable; in other use cases, this might return a completely different object instead.

When run, this module's self-test code uses its context manager to trace the entry and exit of two `with` statement blocks. The net effect automatically invokes the manager's `__enter__` and `__exit__` methods:

```
$ python3 withas.py
[starting with block]
running test 1
reached
[exited normally]

[starting with block]
running test 2
[propagating exception: <class 'TypeError'>]
Traceback (most recent call last):
  File "/…/LP6E/Chapter34/withas.py", line 25, in <modu
    raise TypeError
TypeError
```

Context managers can also utilize OOP state information and inheritance, but are somewhat advanced devices meant for tool builders, so we'll skip additional details here. See Python's standard manuals for the full story—including its coverage of the `contextlib` standard module that provides additional tools for coding context managers.

Also remember that the `try` / `finally` combination provides support for termination-time activities too, and is generally sufficient in roles that don't warrant coding classes to support the `with` statement's protocol.

Multiple Context Managers

The `with` statement has one last card to turn over: it may also specify *multiple* (sometimes called "nested") context managers with comma syntax. For example, in the following code snippet that selects lines by substring, both files' exit actions are automatically run to close the files when the statement block exits, regardless of exception outcomes:

```
with open('lines.txt') as input, open('matches.txt', 'w
    for line in input:
        if 'somekey' in line:
            output.write(line)
```

Any number of context manager items may be listed, and multiple items work the same as nested `with` statements. That is, the following hypothetical code:

```
with A() as a, B() as b:
    statements
```

is equivalent to (and possibly simpler than) the following:

```
with A() as a:
    with B() as b:
        statements
```

The net effect is that each context manager's entry and exit method is run in turn on block entry and exit, and exceptions in the block are caught automatically and possibly reraised on `with` exit at the discretion of the outermost manager's exit method. Multiple *file* context managers, for instance, will all be run to open files on entry, and close them on exit—exception or not.

The Termination-Handlers Shoot-Out

You can find more info on context managers in Python's docs. Rather than getting more detailed here, let's close out this chapter with a quick look at this extension in action and a vetting of its roles. Using newer and redundant tools like `with` doesn't in and of itself prove intelligence, and it's important to understand the trade-offs such options imply.

First up, the following codes a parallel *lines scan* of files located in this book's examples package. It uses `with` to open two files at once and then reads and zips together their next-line pairs on each iteration of a `for` loop. Thanks to the file object's context manager, there's no need to manually catch exceptions or close files when finished:

```
>>> with open('lines1.txt') as file1, open('lines2.txt'
        for pair in zip(file1, file2):
            print(pair)

('hack\n', 'HACK\n')
('code\n', 'GOOD\n')
('well\n', 'CODE\n')
```

You might also use this coding structure to do a *lines comparison* of two text files. The following simply replaces the former's `print` with an `if` for a comparison operation, and adds an `enumerate` for automatic line numbers:

```
>>> with open('lines1.txt') as file1, open('lines2.txt'
        for (linenum, (line1, line2)) in enumerate(zip(
            if line1.lower() != line2.lower():
                print(f'{linenum} => {line1!r} != {line

1 => 'code\n' != 'GOOD\n'
2 => 'well\n' != 'CODE\n'
```

That said, `with` isn't all that useful in the preceding examples when using CPython, because *input* file objects don't require a buffer flush, and file objects are *closed* automatically when garbage collected if still open. Moreover, *exceptions* in the `with` block are still propagated outside the statement if not explicitly caught. Hence, the temporary files would be auto-

closed immediately and exception behavior would be the same for simpler code like this:

```
for pair in zip(open('lines1.txt'), open('lines2.txt'))
    print(pair)
```

On the other hand, some of the alternative *Pythons* of Chapter 2 may use different garbage collectors that require direct closes, to avoid taxing system resources. In addition, *output* files may require closes to ensure that any buffered content is transferred to disk so it's available for opens in later code.

The following *lines filter* code addresses both concerns, by automatically closing files on statement exit, exception or not (it also uses parentheses and line splits after `with`, available as of Python 3.10, and omits write counts for brevity):

```
>>> with (open('lines1.txt') as input,
          open('uppers.txt', 'w') as output):
        for line in input:
            output.write(line.upper())

>>> print(open('uppers.txt').read())          # File cc
HACK
CODE
WELL
```

Still, in simple scripts, we can often just open files in separate statements and close after processing if needed. There's no point in catching an exception if it means your program is out of business anyhow, and closes are required to flush output buffers only if files will be reopened by later code—which may never be reached after exceptions anyhow:

```
input  = open('lines1.txt')
output = open('uppers.txt', 'w')
for line in input:                             # Same ej
    output.write(line.upper())                 # and fil
```

Nevertheless, in programs that must *both* continue after exceptions and close output files for later use in the same program (or REPL) run, the `with` avoids an equivalent `try` / `finally` combination that may be more obvious to some readers, but also requires noticeably more code—eight lines instead of four, quantitatively speaking:

```
input  = open('lines1.txt')
output = open('uppers.txt', 'w')
try:                                        # Same but
    for line in input:
        output.write(line.upper())
finally:
    input.close()                           # Ensure c
    output.close()                          # Whether
```

Even so, the `try` / `finally` is a single tool that applies to all finalization cases and makes code explicit. The `with` can be more concise for context-manager users, but applies only to objects that implement its complex protocol, relies on implicit "magic" that obscures meaning, and adds redundancy that doubles the required knowledge base of programmers. As usual, you'll have to weigh these tools' trade-offs for yourself.

Chapter Summary

In this chapter, we took a more detailed look at exception processing by exploring the statements related to exceptions in Python: `try` to catch them, `raise` to trigger them, `assert` to raise them conditionally, and `with` to wrap code blocks in context managers that automate entry and exit actions.

Up to this point, exceptions may seem like a fairly lightweight tool (apart from the `with` protocol, that is). The most complex thing about them may be how they are identified—a topic the next chapter will address by showing how exception objects are made. As you'll see there, classes allow you to code new exceptions specific to your programs. Before we move ahead, though, let's work through the following short quiz on the basics covered here.

Test Your Knowledge: Quiz

1. What is the `try` statement for?
2. What are the two common variations of the `try` statement?
3. What is the `raise` statement for?
4. What is the `assert` statement designed to do, and what other statement is it like?
5. What is the `with` statement designed to do, and what other statement is it like?

Test Your Knowledge: Answers

1. The `try` statement catches and recovers from exceptions—it specifies a block of code to run and one or more handlers for exceptions that may be raised during the block's execution.
2. The two common variations on the `try` statement are `try` / `except` / `else` (for catching exceptions) and `try` / `finally` (for specifying cleanup actions that must occur whether an exception is raised or not). Despite these logically distinct roles, the `except` and `finally` blocks may be mixed in the same statement, so the two forms are really part of the single `try` statement. Even when mixed with `except`, though, the `finally` is still run on the way out of the `try`, regardless of what exceptions may have been raised or handled. In fact, the combined form is equivalent to nesting a `try` / `except` / `else` in a `try` / `finally`.
3. The `raise` statement raises (triggers) an exception. Python raises built-in exceptions on errors internally, but your scripts can trigger built-in or user-defined exceptions too with `raise`.
4. The `assert` statement raises an `AssertionError` exception if a condition is false. It's similar to a conditional `raise` statement wrapped up in an `if` statement, and can be disabled with a `-O` command switch.
5. The `with` statement is designed to automate startup and termination activities that must occur around a block of code. It is roughly like a `try` / `finally` combination in that its exit actions run whether an exception occurred or not, but it employs an object-based protocol for specifying entry and exit actions and may reduce code size for context-manger users. Still, it's not quite as general, as it applies only to objects

that support its protocol; `try` with `finally` clauses can handle more use cases.