5
Control Flow

This chapter describes statements related to the control flow of a program. Topics include conditionals, loops, and exceptions.

Conditionals

The `if`, `else`, and `elif` statements control conditional code execution. The general format of a conditional statement is as follows:

```
if expression:
    statements
elif expression:
    statements
elif expression:
    statements
...
else:
    statements
```

If no action is to be taken, you can omit both the `else` and `elif` clauses of a conditional. Use the `pass` statement if no statements exist for a particular clause:

```
if expression:
    pass            # Do nothing
else:
    statements
```

Loops

You implement loops using the `for` and `while` statements. For example:

```
while expression:
    statements

for i in s:
    statements
```

The `while` statement executes statements until the associated expression evaluates to false. The `for` statement iterates over all the elements in a sequence until no more elements are available. If the elements of the sequence are tuples of identical size, you can use the following variation of the `for` statement:

```
for x,y,z in s:
    statements
```

In this case, `s` must be a sequence of tuples, each with three elements. On each iteration, the contents of the variables `x`, `y`, and `z` are assigned the contents of the corresponding tuple.

To break out of a loop, use the `break` statement. For example, the following function reads lines of text from the user until an empty line of text is entered:

```
while 1:
   cmd = raw_input('Enter command > ')
   if not cmd:
      break            # No input, stop loop
   # process the command
    ...
```

To jump to the next iteration of a loop (skipping the remainder of the loop body), use the `continue` statement. This statement tends to be used less often, but is sometimes useful when the process of reversing a test and indenting another level would make the program too deeply nested or unnecessarily complicated. As an example, the following loop prints only the non-negative elements of a list:

```
for a in s:
    if a < 0:
       continue       # Skip negative elements
    print a
```

The `break` and `continue` statements apply only to the innermost loop being executed. If it's necessary to break out of a deeply nested loop structure, you can use an exception. Python doesn't provide a goto statement.

You can also attach the `else` statement to loop constructs, as in the following example:

```
# while-else
while i < 10:
     do something
     i = i + 1
else:
     print 'Done'

# for-else
for a in s:
     if a == 'Foo':
          break
else:
     print 'Not found!'
```

The `else` clause of a loop executes only if the loop runs to completion. This either occurs immediately (if the loop wouldn't execute at all) or after the last iteration. On the other hand, if the loop is terminated early using the `break` statement, the `else` clause is skipped.

Exceptions

Exceptions indicate errors and break out of the normal control flow of a program. An exception is raised using the `raise` statement. The general format of the `raise` statement is `raise` *`exception`* `[,` *`value`*`]` where *`exception`* is the exception type and *`value`* is an optional value giving specific details about the exception. For example:

```
raise RuntimeError, 'Unrecoverable Error'
```

If the `raise` statement is used without any arguments, the last exception generated is raised again (although this works only while handling a previously raised exception).

To catch an exception, use the `try` and `except` statements, as shown here:

```
try:
    f = open('foo')
except IOError, e:
    print "Unable to open 'foo': ", e
```

When an exception occurs, the interpreter stops executing statements in the `try` block and looks for an `except` clause that matches the exception that has occurred. If found, control is passed to the first statement in the `except` clause. Otherwise, the exception is propagated up to the block of code in which the `try` statement appeared. This code may itself be enclosed in a `try-except` that can handle the exception. If an exception works its way up to the top level of a program without being caught, the interpreter aborts with an error message. If desired, uncaught exceptions can also be passed to a user-defined function `sys.excepthook()` as described in Appendix A, "The Python Library," `sys` module.

The optional second argument to the `except` statement is the name of a variable in which the argument supplied to the `raise` statement is placed if an exception occurs. Exception handlers can examine this value to find out more about the cause of the exception.

Multiple exception-handling blocks are specified using multiple `except` clauses, such as in the following example:

```
try:
   do something
except IOError, e:
   # Handle I/O error
   ...
except TypeError, e:
   # Handle Type error
   ...
except NameError, e:
   # Handle Name error
   ...
```

A single handler can catch multiple exception types like this:

```
try:
   do something
except (IOError, TypeError, NameError), e:
   # Handle I/O, Type, or Name errors
   ...
```

To ignore an exception, use the `pass` statement as follows:

```
try:
   do something
except IOError:
   pass             # Do nothing (oh well).
```

To catch all exceptions, omit the exception name and value:

```
try:
   do something
except:
   print 'An error occurred'
```

Table 5.1 Built-in Exceptions

Exception	Description
`Exception`	The root of all exceptions
`SystemExit`	Generated by `sys.exit()`
`StandardError`	Base for all built-in exceptions
`ArithmeticError`	Base for arithmetic exceptions
`FloatingPointError`	Failure of a floating-point operation
`OverflowError`	Arithmetic overflow
`ZeroDivisionError`	Division or modulus operation with 0
`AssertionError`	Raised by the `assert` statement
`AttributeError`	Raised when an attribute name is invalid
`EnvironmentError`	Errors that occur externally to Python
`IOError`	I/O or file-related error
`OSError`	Operating system error
`WindowsError`	Error in Windows
`EOFError`	Raised when the end of the file is reached
`ImportError`	Failure of the `import` statement
`KeyboardInterrupt`	Generated by the interrupt key (usually Ctrl+C)
`LookupError`	Indexing and key errors
`IndexError`	Out-of-range sequence offset
`KeyError`	Nonexistent dictionary key
`MemoryError`	Out of memory
`NameError`	Failure to find a local or global name
`UnboundLocalError`	Unbound local variable
`RuntimeError`	A generic catch-all error
`NotImplementedError`	Unimplemented feature
`SyntaxError`	Parsing error
`TabError`	Inconsistent tab usage (generated with `-tt` option)
`IndentationError`	Indentation error
`SystemError`	Nonfatal system error in the interpreter
`TypeError`	Passing an inappropriate type to an operation
`ValueError`	Inappropriate or missing value
`UnicodeError`	Unicode encoding error

The `try` statement also supports an `else` clause, which must follow the last `except` clause. This code is executed if the code in the `try` block doesn't raise an exception. Here's an example:

```
try:
   f = open('foo', 'r')
except IOError:
   print 'Unable to open foo'
else:
   data = f.read()
   f.close()
```

The `finally` statement defines a cleanup action for code contained in a `try` block. For example:

```
f = open('foo','r')
try:
   # Do some stuff
   ...
finally:
   f.close()
   print "File closed regardless of what happened."
```

The `finally` clause isn't used to catch errors. Rather, it's used to provide code that must always be executed, regardless of whether an error occurs. If no exception is raised, the code in the `finally` clause is executed immediately after the code in the `try` block. If an exception occurs, control is first passed to the first statement of the `finally` clause. After this code has executed, the exception is re-raised to be caught by another exception handler. The `finally` and `except` statements cannot appear together within a single `try` statement.

Python defines the built-in exceptions listed in Table 5.1. (For specific details about these exceptions, see Appendix A.)

All the exceptions in a particular group can be caught by specifying the group name in an `except` clause. For example,

```
try:
     statements
except LookupError:     # Catch IndexError or KeyError
     statements
```

or

```
try:
     statements
except StandardError:   # Catch any built-in exception
     statements
```

Defining New Exceptions

All the built-in exceptions are defined in terms of classes. To create a new exception, create a new class definition that inherits from `exceptions.Exception` such as the following:

```
import exceptions
# Exception class
class NetworkError(exceptions.Exception):
     def __init__(self,args=None):
         self.args = args
```

The name `args` should be used as shown. This allows the value used in the `raise` statement to be properly printed in tracebacks and other diagnostics. In other words,

```
raise NetworkError, "Cannot find host."
```

creates an instance of `NetworkError` using the call

```
NetworkError("Cannot find host.")
```

The object that is created will print itself as `NetworkError: Cannot find host.` If you use a name other than `self.args` or you don't store the argument, this feature won't work correctly.

When an exception is raised, the optional value supplied in the `raise` statement is used as the argument to the exception's class constructor. If the constructor for an exception requires more than one argument, it can be raised in two ways:

```
import exceptions
# Exception class
class NetworkError(exceptions.Exception):
     def __init__(self,errno,msg):
         self.args = (errno, msg)
         self.errno = errno
         self.errmsg = msg

# Raises an exception (multiple arguments)
def error2():
     raise NetworkError(1, 'Host not found')

# Raises an exception (multiple arguments)
def error3():
     raise NetworkError, (1, 'Host not found')
```

Class-based exceptions enable you to create hierarchies of exceptions. For instance, the `NetworkError` exception defined earlier could serve as a base class for a variety of more specific errors. For example:

```
class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

def error3():
    raise HostnameError

def error4():
    raise TimeoutError

try:
    error3()
except NetworkError:
    import sys
    print sys.exc_type    # Prints exception type
```

In this case, the `except NetworkError` statement catches any exception derived from `NetworkError`. To find the specific type of error that was raised, examine the variable `sys.exc_type`. Similarly, the `sys.exc_value` variable contains the value of the last exception. Alternatively, the `sys.exc_info()` function can be used to retrieve exception information in a manner that doesn't rely on global variables and is thread-safe.

Assertions and __debug__

The `assert` statement is used to add debugging code into a program. The general form of `assert` is

```
assert test [, data]
```

where *`test`* is an expression that should evaluate to true or false. If *`test`* evaluates to false, `assert` raises an `AssertionError` exception with the optional *`data`* supplied to the `assert` statement. For example:

```
def write_data(file,data):
    assert file, "write_data: file is None!"
    ...
```

Internally, the `assert` statement is translated into the following code:

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

`__debug__` is a built-in read-only value that's set to `1` unless the interpreter is running in optimized mode (specified with the `-O` option). Although `__debug__` is used by assertions, you also can use it to include any sort of debugging code.

The `assert` statement should not be used for code that must be executed to make the program correct, since it won't be executed if Python is run in optimized mode. In particular, it's an error to use `assert` to check user input. Instead, `assert` statements are used to check things that should always be true; if one is violated, it represents a bug in the program, not an error by the user.

For example, if the function `write_data()` shown here were intended for use by an end user, the `assert` statement should be replaced by a conventional `if` statement and the desired error handling.