Exception Handling and Debugging

Any good program makes use of a language's exception handling mechanisms. There is no better way to frustrate an end-user then by having them run into an issue with your software and displaying a big ugly error message on the screen, followed by a program crash. Exception handling is all about ensuring that when your program encounters an issue, it will continue to run and provide informative feedback to the end-user or program administrator. Any Java programmer becomes familiar with exception handling on day one, as some Java code won't even compile unless there is some form of exception handling put into place via the try-catch-finally syntax. Python has similar constructs to that of Java, and we'll discuss them in this chapter.

After you have found an exception, or preferably before your software is distributed, you should go through the code and debug it in order to find and repair the erroneous code. There are many different ways to debug and repair code; we will go through some debugging methodologies in this chapter. In Python as well as Java, the assert keyword can help out tremendously in this area. We'll cover assert in depth here and learn the different ways that it can be used to help you out and save time debugging those hard-to-find errors.

Exception Handling Syntax and Differences with Java

Java developers are very familiar with the try-catch-finally block as this is the main mechanism that is used to perform exception handling. Python exception handling differs a bit from Java, but the syntax is fairly similar. However, Java differs a bit in the way that an exception is thrown in code. Now, realize that I just used the term throw…this is Java terminology. Python does not throw exceptions, but instead it raises them. Two different terms which mean basically the same thing. In this section, we'll step through the process of handling and raising exceptions in Python code, and show you how it differs from that in Java.

For those who are unfamiliar, I will show you how to perform some exception handling in the Java language. This will give you an opportunity to compare the two syntaxes and appreciate the flexibility that Python offers.

Listing 7-1. Exception Handling in Java

```
try {
// perform some tasks that may throw an exception
} catch (ExceptionType messageVariable) {
// perform some exception handling
} finally {
// execute code that must always be invoked
}
```

Now let's go on to learn how to make this work in Python. Not only will we see how to handle and raise exceptions, but you'll also learn some other great techniques such as using assertions later in the chapter.

Catching Exceptions

How often have you been working in a program and performed some action that caused the program to abort and display a nasty error message? It happens more often than it should because most exceptions can be caught and handled nicely. By nicely, I mean that the program will not abort and the end user will receive a descriptive error message stating what the problem is, and in some cases how it can be resolved. The exception handling mechanisms within programming languages were developed for this purpose.

Listing 7-2. try-except Example

```
# This function uses a try-except clause to provide a nice error
# message if the user passes a zero in as the divisor
>>> from __future__ import division
>>> def divide_numbers(x, y):
...     try:
...         return x/y
...     except ZeroDivisionError:
...         return 'You cannot divide by zero, try again'
…
# Attempt to divide 8 by 3
>>> divide_numbers(8,3)
2.6666666666666665
# Attempt to divide 8 by zero
>>> divide_numbers(8, 0)
'You cannot divide by zero, try again'
```

Table 7-1 lists of all exceptions that are built into the Python language along with a description of each. You can write any of these into an except clause and try to handle them. Later in this chapter I will show you how you and raise them if you'd like. Lastly, if there is a specific type of exception that you'd like to throw that does not fit any of these, then you can write your own exception type object. It is important to note that Python exception handling differs a bit from Java exception handling. In Java, many times the compiler forces you to catch exceptions, such is known as checked exceptions. Checked exceptions are basically exceptions that a method may throw while performing some task. The developer is forced to handle these checked exceptions using a try/catch or a throws clause, otherwise the compiler complains. Python has no such facility built into its error handling system. The developer decides when to handle exceptions and when not to do so. It is a best practice to include error handling wherever possible even though the interpreter does not force it.

Exceptions in Python are special classes that are built into the language. As such, there is a class hierarchy for exceptions and some exceptions are actually subclasses of another exception class. In this case, a program can handle the superclass of such an exception and all

subclassed exceptions are handled automatically. Table 7-1 lists the exceptions defined in the Python language, and the indentation resembles the class hierarchy.

Table 7-1. Exceptions

Exception	Description
BaseException	This is the root exception for all others
GeneratorExit	Raised by close() method of generators for terminating iteration
KeyboardInterrupt	Raised by the interrupt key
SystemExit	Program exit
Exception	Root for all non-exiting exceptions
StopIteration	Raised to stop an iteration action
StandardError	Base class for all built-in exceptions
ArithmeticError	Base for all arithmetic exceptions
FloatingPointError	Raised when a floating-point operation fails
OverflowError	Arithmetic operations that are too large
ZeroDivisionError	Division or modulo operation with zero as divisor
AssertionError	Raised when an assert statement fails
AttributeError	Attribute reference or assignment failure
EnvironmentError	An error occurred outside of Python
IOError	Error in Input/Output operation
OSError	An error occurred in the os module
EOFError	input() or raw_input() tried to read past the end of a file
ImportError	Import failed to find module or name
LookupError	Base class for IndexError and KeyError
IndexError	A sequence index goes out of range
KeyError	Referenced a non-existent mapping (dict) key
MemoryError	Memory exhausted
NameError	Failure to find a local or global name
UnboundLocalError	Unassigned local variable is referenced
ReferenceError	Attempt to access a garbage-collected object
RuntimeError	Obsolete catch-all error
NotImplementedError	Raised when a feature is not implemented
SyntaxError	Parser encountered a syntax error
IndentationError	Parser encountered an indentation issue
TabError	Incorrect mixture of tabs and spaces
SystemError	Non-fatal interpreter error
TypeError	Inappropriate type was passed to an operator or function
ValueError	Argument error not covered by TypeError or a more precise error
Warning	Base for all warnings

The try-except-finally block is used in Python programs to perform the exception-handling task. Much like that of Java, code that may or may not raise an exception can be placed in the try block. Differently though, exceptions that may be caught go into an except block much like the Java catch equivalent. Any tasks that must be performed no matter if an exception is thrown or not should go into the finally block. All tasks within the finally block are performed if an exception is raised either within the except block or by some other exception. The tasks are also performed before the exception is raised to ensure that they are completed. The finally block is a great place to perform cleanup activity such as closing open files and such.

Listing 7-3. try-except-finally Logic

```
try:
    # perform some task that may raise an exception
except Exception, value:
    # perform some exception handling
finally:
    # perform tasks that must always be completed (Will be performed before
the exception is # raised.)
```

Python also offers an optional else clause to create the try-except-else logic. This optional code placed inside the else block is run if there are no exceptions found in the block.

Listing 7-4. try-finally logic

```
try:
    # perform some tasks that may raise an exception
finally:
    # perform tasks that must always be completed (Will be performed before
the exception is # raised.)
```

The *else* clause can be used with the exception handling logic to ensure that some tasks are only run if no exceptions are raised. Code within the else clause is only initiated if no exceptions are thrown, and if any exceptions are raised within the else clause the control does not go back out to the except. Such activities to place in inside an else clause would be transactions such as a database commit. If several database transactions were taking place inside the try clause you may not want a commit to occur unless there were no exceptions raised.

Listing 7-5. try-except-else logic:

```
try:
    # perform some tasks that may raise an exception
except:
    # perform some exception handling
else:
    # perform some tasks thatwill only be performed if no exceptions are
thrown
```

You can name the specific type of exception to catch within the except block, or you can generically define an exception handling block by not naming any exception at all. Best practice of course states that you should always try to name the exception and then provide the best possible handling solution for the case. After all, if the program is simply going to spit out a nasty error then the exception handling block is not very user friendly and is only helpful to developers. However, there are some rare cases where it would be advantageous to not explicitly refer to an exception type when we simply wish to ignore errors and move on. The except block also allows us to define a variable to which the exception message will be assigned. This allows us the ability to store that message and display it somewhere within our exception handling code block. If you are calling a piece of Java code from within Jython and the Java code throws an exception, it can be handled within Jython in the same manner as Jython exceptions.

Listing 7-6. Exception Handling in Python

```
# Code without an exception handler
>>> x = 10
>>> z = x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined
# The same code with an exception handling block
>>> x = 10
>>> try:
...     z = x / y
... except NameError, err:
...     print "One of the variables was undefined: ", err
...
One of the variables was undefined:  name 'y' is not defined
```

It is important to note that Jython 2.5.x uses the Python 2.5.x exception handling syntax. This syntax will be changing in future releases of Jython. Take note of the syntax that is being used for defining the variable that holds the exception. Namely, the except ExceptionType, value statement syntax in Python and Jython 2.5 differs from that beyond 2.5. In Python 2.6, the syntax changes a bit in order to ready developers for Python 3, which exclusively uses the new syntax.

Listing 7-7. Jython and Python 2.5 and Prior

```
try:
    # code
except ExceptionType, messageVar:
    # code
```

Listing 7-8. Jython 2.6 (Not Yet Implemented) and Python 2.6 and Beyond

```
try:
   # code
```

```
except ExceptionType as messageVar:
    # code
```

We had previously mentioned that it was simply bad programming practice to not explicitly name an exception type when writing exception handling code. This is true, however Python provides us with another a couple of means to obtain the type of exception that was thrown. The easiest way to find an exception type is to simply catch the exception as a variable as we've discussed previously. You can then find the specific exception type by using the type(error_variable) syntax if needed.

Listing 7-9. Determining Exception Type

```
# In this example, we catch a general exception and then determine the type
later
>>> try:
...     8/0
... except Exception, ex1:
...     'An error has occurred'
...
'An error has occurred'
>>> ex1
ZeroDivisionError('integer division or modulo by zero',)
>>> type(ex1)
<type 'exceptions.ZeroDivisionError'>
>>>
```

There is also a function provided in the sys package known as sys.exc_info() that will provide us with both the exception type and the exception message. This can be quite useful if we are wrapping some code in a try-except block but we really aren't sure what type of exception may be thrown. Below is an example of using this technique.

Listing 7-10. Using sys.exc_info()

```
# Perform exception handling without explicitly naming the exception type
>>> x = 10
>>> try:
...     z = x / y
... except:
...     print "Unexpected error: ", sys.exc_info()[0], sys.exc_info()[1]
...
Unexpected error:  <type 'exceptions.NameError'> name 'y' is not defined
```

Sometimes you may run into a situation where it is applicable to catch more than one exception. Python offers a couple of different options if you need to do such exception handling. You can either use multiple except clauses, which does the trick and works well if you're interested in performing different tasks for each different exception that occurs, but may become too wordy. The other preferred option is to enclose your exception types within parentheses and separated

by commas on your except statement. Take a look at the following example that portrays the latter approach using Listing 7-6.

Listing 7-11. Handling Multiple Exceptions

```
# Catch NameError, but also a ZeroDivisionError in case a zero is used in the
equation
>>> try:
...     z = x/y
... except(NameError, ZeroDivisionError), err:
...     "An error has occurred, please check your values and try again"
...
'An error has occurred, please check your values and try again'

# Using multiple except clauses
>>> x = 10
>>> y = 0
>>> try:
...     z = x / y
... except NameError, err1:
...     print err1
... except ZeroDivisionError, err2:
...     print 'You cannot divide a number by zero!'
...
You cannot divide a number by zero!
```

As mentioned previously, an exception is simply a class in Python. There are superclasses and subclasses for exceptions. You can catch a superclass exception to catch any of the exceptions that subclass that exception are thrown. For instance, if a program had a specific function that accepted either a list or dict object, it would make sense to catch a LookupError as opposed to finding a KeyError or IndexError separately. Look at the following example to see one way that this can be done.

Listing 7-12. Catching a Superclass Exceptions

```
# In the following example, we define a function that will return
# a value from some container.  The function accepts either lists
# or dictionary objects.  The LookupError superclass is caught
# as opposed to checking for each of it's subclasses...namely KeyError and
IndexError.
>>> def find_value(obj, value):
...     try:
...         return obj[value]
...     except LookupError, ex:
...         return 'An exception has been raised, check your values and try
again'
...

# Create both a dict and a list and test the function by looking for a value
that does
# not exist in either container
>>> mydict = {'test1':1,'test2':2}
```

```
>>> mylist = [1,2,3]
>>> find_value(mydict, 'test3')
'An exception has been raised, check your values and try again'
>>> find_value(mylist, 2)
3
>>> find_value(mylist, 3)
'An exception has been raised, check your values and try again'
>>>
```

If multiple exception blocks have been coded, the first matching exception is the one that is caught. For instance, if we were to redesign the find_value function that was defined in the previous example, but instead raised each exception separately then the first matching exception would be raised. . .the others would be ignored. Let's see how this would work.

Listing 7-13. Catching the First Matching Exceptions

```
# Redefine the find_value() function to check for each exception separately
# Only the first matching exception will be raised, others will be ignored.
# So in these examples, the except LookupError code is never run.
>>> def find_value(obj, value):
...     try:
...        return obj[value]
...     except KeyError:
...         return 'The specified key was not in the dict, please try again'
...     except IndexError:
...         return 'The specified index was out of range, please try again'
...     except LookupError:
...         return 'The specified key was not found, please try again'
...
>>> find_value(mydict, 'test3')
'The specified key was not in the dict, please try again'
>>> find value(mylist, 3)
'The specified index was out of range, please try again'
>>>
```

The try-except block can be nested as deep as you'd like. In the case of nested exception handling blocks, if an exception is thrown then the program control will jump out of the inner most block that received the error, and up to the block just above it. This is very much the same type of action that is taken when you are working in a nested loop and then run into a break statement, your code will stop executing and jump back up to the outer loop. The following example shows an example for such logic.

Listing 7-14. Nested Exception Handling Blocks

```
# Perform some division on numbers entered by keyboard
try:
    # do some work
    try:
        x = raw_input ('Enter a number for the dividend:  ')
        y = raw_input('Enter a number to divisor: ')
        x = int(x)
```

```
        y = int(y)
    except ValueError:
        # handle exception and move to outer try-except
        print 'You must enter a numeric value!'
    z = x / y
except ZeroDivisionError:
    # handle exception
    print 'You cannot divide by zero!'
except TypeError:
    print 'Retry and only use numeric values this time!'
else:
    print 'Your quotient is: %d' % (z)
```

In the previous example, we nested the different exception blocks. If the first ValueError were raised, it would give control back to the outer exception block. Therefore, the ZeroDivisionError and TypeError could still be raised. Otherwise, if those last two exceptions are not thrown then the tasks within the else clause would be run.

As stated previously, it is a common practice in Jython to handle Java exceptions. Oftentimes we have a Java class that throws exceptions, and these can be handled or displayed in Jython just the same way as handling Python exceptions.

Listing 7-15. Handling Java Exceptions in Jython

```
// Java Class TaxCalc
public class TaxCalc {
    public static void main(String[] args) {
    double cost = 0.0;
    int pct   = 0;
    double tip = 0.0;
    try {
        cost = Double.parseDouble(args[0]);
        pct = Integer.parseInt(args[1]);
        tip = (cost * (pct * .01));
        System.out.println("The total gratutity based on " + pct + " percent
would be " + tip);
        System.out.println("The total bill would be " + (cost + tip) );
    } catch (NumberFormatException ex){
        System.out.println("You must pass number values as arguments.
 Exception: " + ex);
    } catch (ArrayIndexOutOfBoundsException ex1){
        System.out.println("You must pass two values to this utility.  " +
        "Format: TaxCalc(cost, percentage)  Exception: " + ex1);
    }
    }
}
```

Using Jython:

```
# Now lets bring the TaxCalc Java class into Jython and use it
```

```
>>> import TaxCalc
>>> calc = TaxCalc()

# pass strings within a list to the TaxCalc utility and the Java exception
will be thrown
>>> vals = ['test1','test2']
>>> calc.main(vals)
You must pass number values as arguments.  Exception:
java.lang.NumberFormatException: For input string: "test1"

# Now pass numeric values as strings in a list, this works as expected
(except for the bad
# rounding)
>>> vals = ['25.25', '20']
>>> calc.main(vals)
The total gratutity based on 20 percent would be 5.050000000000001
The total bill would be 30.3
```

You can also throw Java exceptions in Jython by simply importing them first and then using then raising them just like Python exceptions.

Raising Exceptions

Often you will find reason to raise your own exceptions. Maybe you are expecting a certain type of keyboard entry, and a user enters something incorrectly that your program does not like. This would be a case when you'd like to raise your own exception. The raise statement can be used to allow you to raise an exception where you deem appropriate. Using the raise statement, you can cause any of the Python exception types to be raised, you could raise your own exception that you define (discussed in the next section). The raise statement is analogous to the throw statement in the Java language. In Java we may opt to throw an exception if necessary. However, Java also allows you to apply a throws clause to a particular method if an exception may possibly be thrown within instead of using try-catch handler in the method. Python does not allow you do perform such techniques using the raise statement.

Listing 7-16. raise Statement Syntax

```
raise ExceptionType or String[, message[, traceback]]
```

As you can see from the syntax, using raise allows you to become creative in that you could use your own string when raising an error. However, this is not really looked upon as a best practice as you should try to raise a defined exception type if at all possible. You can also provide a short message explaining the error. This message can be any string. Let's take a look at an example.

Listing 7-17. raising Exceptions Using Message

```
>>> raise Exception("An exception is being raised")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
Exception: An exception is being raised
>>> raise TypeError("You've specified an incorrect type")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: You've specified an incorrect type
```

Now you've surely seen some exceptions raised in the Python interpreter by now. Each time an exception is raised, a message appears that was created by the interpreter to give you feedback about the exception and where the offending line of code may be. There is always a traceback section when any exception is raised. This really gives you more information on where the exception was raised. Lastly, let's take a look at raising an exception using a different format. Namely, we can use the format raise Exception, "message".

Listing 7-18. Using the raise Statement with the Exception, "message" Syntax

```
>>> raise TypeError,"This is a special message"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: This is a special message
```

Defining Your Own Exceptions

You can define your own exceptions in Python by creating an exception class. You simply define a class that inherits from the base Exception class. The easiest defined exception can simply use a pass statement inside the class. Exception classes can accept parameters using the initializer, and return the exception using the __str__ method. Any exception you write should accept a message. It is also a good practice to name your exception giving it a suffix of Error if the exception is referring to an error of some kind.

Listing 7-19. Defining a Basic Exception Class

```
class MyNewError(Exception):
    pass
```

This example is the simplest type of exception you can create. This exception that was created above can be raised just like any other exception now.

```
raise MyNewError("Something happened in my program")
```

A more involved exception class may be written as follows.

Listing 7-20. Exception Class Using Initializer

```
class MegaError(Exception):
    """ This is raised when there is a huge problem with my program"""
```

```
    def __init__(self, val):
        self.val = val
    def __str__(self):
        return repr(self.val)
```

Issuing Warnings

Warnings can be raised at any time in your program and can be used to display some type of warning message, but they do not necessarily cause execution to abort. A good example is when you wish to deprecate a method or implementation but still make it usable for compatibility. You could create a warning to alert the user and let them know that such methods are deprecated and point them to the new definition, but the program would not abort. Warnings are easy to define, but they can be complex if you wish to define rules on them using filters. Warning filters are used to modify the behavior of a particular warning. Much like exceptions, there are a number of defined warnings that can be used for categorizing. In order to allow these warnings to be easily converted into exceptions, they are all instances of the Exception type. Remember that exceptions are not necessarily errors, but rather alerts or messages. For instance, the StopIteration exception is raised by a program to stop the iteration of a loop…not to flag an error with the program.

To issue a warning, you must first import the warnings module into your program. Once this has been done then it is as simple as making a call to the warnings.warn() function and passing it a string with the warning message. However, if you'd like to control the type of warning that is issued, you can also pass the warning class. Warnings are listed in Table 7-2.

Listing 7-21. Issuing a Warning

```
# Always import the warnings module first
import warnings

# A couple of examples for setting up warnings
warnings.warn("this feature will be deprecated")
warnings.warn("this is a more involved warning", RuntimeWarning)

# Using A Warning in a Function

# Suppose that use of the following function has been deprecated,
# warnings can be used to alert the function users

# The following function calculates what the year will be if we
# add the specified number of days to the current year.  Of course,
# this is pre-Y2K code so it is being deprecated.  We certainly do not
# want this code around when we get to year 3000!
>>> def add_days(current_year, days):
...     warnings.warn("This function has been deprecated as of version x.x",
DeprecationWarning)
...     num_years = 0
...     if days > 365:
...         num_years = days/365
...     return current_year + num_years
```

```
...

# Calling the function will return the warning that has been set up,
# but it does not raise an error...the expected result is still returned.
>>> add_days(2009, 450)
__main__:2: DeprecationWarning: This function has been deprecated as of
version x.x
2010
```

Table 7-2. Python Warning Categories

Warning	Description
Warning	Root warning class
UserWarning	A user-defined warning
DeprecationWarning	Warns about use of a deprecated feature
SyntaxWarning	Syntax issues
RuntimeWarning	Runtime issues
FutureWarning	Warns that a particular feature will be changing in a future release

Importing the warnings module into your code gives you access to a number of built-in warning functions that can be used. If you'd like to filter a warning and change its behavior then you can do so by creating a filter. Table 7-3 lists functions that come with the warnings module.

Table 7-3. Warning Functions

Function	Description
warn(message[, category[, stacklevel]])	Issues a warning. Parameters include a message string, the optional category of warning, and the optional stack level that tells which stack frame the warning should originate from, usually either the calling function or the source of the function itself.
warn_explicit(message, category, filename, lineno[, module[, registry]])	This offers a more detailed warning message and makes category a mandatory parameter. filename, lineno, and module tell where the warning is located. registry represents all of the current warning filters that are active.
showwarning(message, category, filename, lineno[, file])	Gives you the ability to write the warning to a file.
formatwarning(message, category, filename, lineno)	Creates a formatted string representing the warning.
simplefilter(action[, category[, lineno[, append]]])	Inserts simple entry into the ordered list of warnings filters. Regular expressions are not needed for simplefilter as the filter always matches any message in any module as long as the category and line number match. filterwarnings() described below uses a regular expression to match against warnings.

resetwarnings()	Resets all of the warning filters.
filterwarnings(action[, message[, category[, module[, lineno[, append]]]]])	This adds an entry into a warning filter list. Warning filters allow you to modify the behavior of a warning. The action in the warning filter can be one from those listed in Table 7-4, message is a regular expression, category is the type of a warning to be issued, module can be a regular expression, lineno is a line number to match against all lines, append specifies whether the filter should be appended to the list of all filters.

Table 7-4. Python Filter Actions

Filter Actions	
‘always’	Always print warning message
‘default’	Print warning once for each location where warning occurs
‘error’	Converts a warning into an exception
‘ignore’	Ignores the warning
‘module’	Print warning once for each module in which warning occurs
‘once’	Print warning only one time

Let’s take a look at a few ways to use warning filters in the examples below.

Listing 7-22. Warning Filter Examples

```
# Set up a simple warnings filter to raise a warning as an exception

>>> warnings.simplefilter('error', UserWarning)
>>> warnings.warn('This will be raised as an exception')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 63, in
warn
    warn_explicit(message, category, filename, lineno, module, registry,
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 104, in
warn_explicit
    raise message
UserWarning: This will be raised as an exception

# Turn off all active filters using resetwarnings()
>>> warnings.resetwarnings()
>>> warnings.warn('This will not be raised as an exception')
__main__:1: UserWarning: This will not be raised as an exception

# Use a regular expression to filter warnings
# In this case, we ignore all warnings containing the word “one”
>>> warnings.filterwarnings('ignore', '.*one*.',)
>>> warnings.warn('This is warning number zero')
__main__:1: UserWarning: This is warning number zero
>>> warnings.warn('This is warning number one')
>>> warnings.warn('This is warning number two')
```

```
__main__:1: UserWarning: This is warning number two
>>>
```

There can be many different warning filters in use, and each call to the filterwarnings() function will append another warning to the ordered list of filters if so desired. The specific warning is matched against each filter specification in the list in turn until a match is found. In order to see which filters are currently in use, issue the command print warnings.filters. One can also specify a warning filter from the command line by use of the –W option. Lastly, all warnings can be reset to defaults by using the resetwarnings() function.

It is also possible to set up a warnings filter using a command-line argument. This can be quite useful for filtering warnings on a per-script or per-module basis. For instance, if you are interested in filtering warnings on a per-script basis then you could issue the -W command line argument while invoking the script.

Listing 7-23. -W command-line option

```
-Waction:message:category:module:lineno
```

Listing 7-24. Example of using W command line option

```
# Assume we have the following script test_warnings.py
# and we are interested in running it from the command line
import warnings
def test_warnings():
    print "The function has started"
    warnings.warn("This function has been deprecated", DeprecationWarning)
    print "The function has been completed"

if __name__ == "__main__":
    test_warnings()

# Use the following syntax to start and run jython as usual without
# filtering any warnings
jython test_warnings.py
The function has started
test_warnings.py:4: DeprecationWarning: This function has been deprecated
  warnings.warn("This function has been deprecated", DeprecationWarning)
The function has been completed

# Run the script and ignore all deprecation warnings
jython -W "ignore::DeprecationWarning::0" test_warnings.py
The function has started
The function has been completed

# Run the script one last time and treat the DeprecationWarning
# as an exception.  As you see, it never completes
jython -W "error::DeprecationWarning::0" test_warnings.py
The function has started
Traceback (most recent call last):
  File "test_warnings.py", line 8, in <module>
```

```
    test_warnings()
  File "test_warnings.py", line 4, in test_warnings
    warnings.warn("This function has been deprecated", DeprecationWarning)
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 63, in
warn
    warn_explicit(message, category, filename, lineno, module, registry,
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 104, in
warn_explicit
    raise message
DeprecationWarning: This function has been deprecated
```

Warnings can be very useful in some situations. They can be made as simplistic or sophisticated as need be.

Assertions and Debugging

Debugging can be an easy task in Python via use of the assert statement. In CPython, the __debug__ variable can also be used, but this feature is currently not usable in Jython as there is no *optimization* mode for the interpreter. . Assertions are statements that can print to indicate that a particular piece of code is not behaving as expected. The assertion checks an expression for a True or False value, and if it evaluates to False in a Boolean context then it issues an AssertionError along with an optional message. If the expression evaluates to True then the assertion is ignored completely.

```
assert expression [, message]
```

By effectively using the assert statement throughout your program, you can easily catch any errors that may occur and make debugging life much easier. Listing 7-25 will show you the use of the assert statement.

Listing 7-25. Using assert

```
#  The following example shows how assertions are evaluated
>>> x = 5
>>> y = 10
>>> assert x < y, "The assertion is ignored"
>>> assert x > y, "The assertion raises an exception"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: The assertion raises an exception

# Use assertions to validate parameters# Here we check the type of each
parameter to ensure
# that they are integers
>>> def add_numbers(x, y):
...     assert type(x) is int, "The arguments must be integers, please check
the first argument"
...     assert type(y) is int, "The arguments must be integers, please check
the second argument"
...     return x + y
```

```
...
# When using the function, AssertionErrors are raised as necessary
>>> add_numbers(3, 4)
7
>>> add_numbers('hello','goodbye')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add_numbers
AssertionError: The arguments must be integers, please check the first
argument
```

Context Managers

Ensuring that code is written properly in order to manage resources such as files or database connections is an important topic. If files or database connections are opened and never closed then our program could incur issues. Often times, developers elect to make use of the try-finally blocks to ensure that such resources are handled properly. While this is an acceptable method for resource management, it can sometimes be misused and lead to problems when exceptions are raised in programs. For instance, if we are working with a database connection and an exception occurs after we've opened the connection, the program control may break out of the current block and skip all further processing. The connection may never be closed in such a case. That is where the concept of context management becomes an important new feature in Jython. Context management via the use of the with statement is new to Jython 2.5, and it is a very nice way to ensure that resources are managed as expected.

In order to use the with statement, you must import from __future__. The with statement basically allows you to take an object and use it without worrying about resource management. For instance, let's say that we'd like to open a file on the system and read some lines from it. To perform a file operation you first need to open the file, perform any processing or reading of file content, and then close the file to free the resource. Context management using the with statement allows you to simply open the file and work with it in a concise syntax.

Listing 7-26. Python with Statement Example

```
#  Read from a text file named players.txt
>>> from __future__ import with_statement
>>> with open('players.txt','r') as file:
...     x = file.read()
...
>>> print x
Sports Team Management
------------------------------
Josh - forward
Jim - defense
```

In this example, we did not worry about closing the file because the context took care of that for us. This works with object that extends the context management protocol. In other words, any object that implements two methods named *__enter__()* and *__exit__()* adhere to the context

management protocol. When the with statement begins, the *__enter__()* method is executed. Likewise, as the last action performed when the with statement is ending, the*__exit__()* method is executed. The *__enter__()* method takes no arguments, whereas the *__exit__()* method takes three optional arguments type, value, and traceback. The *__exit__()* method returns a True or False value to indicate whether an exception was thrown. The as variable clause on the with statement is optional as it will allow you to make use of the object from within the code block. If you are working with resources such as a lock then you may not need the optional clause.

If you follow the context management protocol, it is possible to create your own objects that can be used with this technique. The *__enter__()* method should create whatever object you are trying to work if needed.

Listing 7-27. Creating a Simple Object That Follows Context Management Protocol

```
# In this example, my_object facilitates the context management protocol
# as it defines an __enter__ and __exit__ method
class my_object:
    def __enter__(self):
        # Perform setup tasks
        return object

    def __exit__(self, type, value, traceback):
        # Perform cleanup
```

If you are working with an immutable object then you'll need to create a copy of that object to work with in the *__enter__()* method. The *__exit__()* method on the other hand can simply return False unless there is some other type of cleanup processing that needs to take place. If an exception is raised somewhere within the context manager, then *__exit__()* is called with three arguments representing type, value, and traceback. However, if there are no exceptions raised then *__exit__()* is passed three None arguments. If *__exit__()* returns True, then any exceptions are "swallowed" or ignored, and execution continues at the next statement after the with-statement.

Summary

In this chapter, we discussed many different topics regarding exceptions and exception handling within a Python application. First, you learned the exception handling syntax of the try-except-finally code block and how it is used. We then discussed why it may be important to raise your own exceptions at times and how to do so. That topic led to the discussion of how to define an exception and we learned that in order to do so we must define a class that extends the Exception type object.

After learning about exceptions, we went into the warnings framework and discussed how to use it. It may be important to use warnings in such cases where code may be deprecated and you want to warn users, but you do not wish to raise any exceptions. That topic was followed by assertions and how assertion statement can be used to help us debug our programs. Lastly, we

touched upon the topic of context managers and using the with statement that is new in Jython 2.5.

Source:
http://www.jython.org/jythonbook/en/1.0/ExceptionHandlingDebug.html