13. Exceptions

13.1. What is an exception?

13.2. Exception Handling Flow-of-control

13.3. Summary

13.4. Standard Exceptions

13.5. Principles for using Exceptions

13.6. Exceptions Syntax

13.7. Glossary

13.8. Exercises

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

13.1. What is an exception?

An exception is a signal that a condition has occurred that can’t be easily handled using the normal flow-of-control of a Python program. Exceptions are often defined as being “errors” but this is not always the case. All errors in Python are dealt with using exceptions, but not all exceptions are errors.

13.2. Exception Handling Flow-of-control

To explain what an exception does, let’s review the normal “flow of control” in a Python program. In normal operation Python executes statements sequentially, one after the other. For three constructs, if-statements, loops and function invocations, this sequential execution is interrupted.

For if-statements, only one of several statement blocks is executed and then flow-of-control jumps to the first statement after the if-statement.

For loops, when the end of the loop is reached, flow-of-control jumps back to the start of the loop and a test is used to determine if the loop needs to execute again. If the loop is finished, flow-of-control jumps to the first statement after the loop.

For function invocations, flow-of-control jumps to the first statement in the called function, the function is executed, and the flow-of-control jumps back to the next statement after the function call.

Do you see the pattern? If the flow-of-control is not purely sequential, it always executes the first statement immediately following the altered flow-of-control. That is why we can say that Python flow-of-control is sequential. But there are cases where this sequential flow-of-control does not work well. An example will best explain this.

Let’s suppose that a program contains complex logic that is appropriately subdivided into functions. The program is running and it currently is executing function D, which was called by function C, which was called by function B, which was called by function A, which was called from the main function. This is illustrated by the following simplistic code example:

def main()

A()

def A():

B()

def B():

C()

def C():

D()

def D()

# processing

Function D determines that the current processing won’t work for some reason and needs to send a message to the main function to try something different. However, all that function D can do using normal flow-of-control is to return a value to function C. So function D returns a special value to function C that means “try something else”. Function C has to recognize this value, quit its processing, and return the special value to function B. And so forth and so on. It would be very helpful if function D could communicate directly with the main function (or functions A and B) without sending a special value through the intermediate calling functions. Well, that is exactly what an exception does. An exception is a message to any function currently on the executing program’s “run-time-stack”. (The “run-time-stack” is what keeps track of the active function calls while a program is executing.)

In Python, your create an exception message using the raise command. The simplest format for a raise command is the keyword raise followed by the name of an exception. For example:

raise ExceptionName

So what happens to an exception message after it is created? The normal flow-of-control of a Python program is interrupted and Python starts looking for any code in its run-time-stack that is interested in dealing with the message. It always searches from its current location at the bottom of the run-time-stack, up the stack, in the order the functions were originally called. A try: except: block is used to say “hey, I can deal with that message.” The first try: except: block that Python finds on its search back up the run-time-stack will be executed. If there is no try: except: block found, the program “crashes” and prints its run-time-stack to the console.

Let’s take a look at several code examples to illustrate this process. If function D had a try: except: block around the code that raised a MyException message, then the flow-of-control would be passed to the local except block. That is, function D would handle it’s own issues.

def main()

A()

def A():

B()

def B():

C()

def C():

D()

def D()

try:

# processing code

if something\_special\_happened:

raise MyException

except MyException:

# execute if the MyException message happened

But perhaps function C is better able to handle the issue, so you could put the try: except: block in function C:

def main()

A()

def A():

B()

def B():

C()

def C():

try:

D()

except MyException:

# execute if the MyException message happened

def D()

# processing code

if something\_special\_happened:

raise MyException

But perhaps the main function is better able to handle the issue, so you could put the try: except: block in the main function:

def main()

try:

A()

except MyException:

# execute if the MyException message happened

def A():

B()

def B():

C()

def C():

D()

def D()

# processing code

if something\_special\_happened:

raise MyException

13.3. Summary

Let’s summarize our discussion. An exception is a message that something “out-of-the-ordinary” has happened and the normal flow-of-control needs to be abandoned. When an exception is raised, Python searches its run-time-stack for a try: except: block that can appropriately deal with the condition. The first try: except: block that knows how to deal with the issue is executed and then flow-of-control is returned to its normal sequential execution. If no appropriate try: except: block is found, the program “crashes” and prints its run-time-stack to the console.

As our final example, here is a program that crashes because no valid try: except: block was found to process the MyException message. Notice that the try: except: block in the main function only knows how to deal with ZeroDivisonError messages, not MyException messages.

def main()

try:

A()

except ZeroDivisonError:

# execute if a ZeroDivisonError message happened

def A():

B()

def B():

C()

def C():

D()

def D()

# processing code

if something\_special\_happened:

raise MyException

13.4. Standard Exceptions

Most of the standard exceptions built into Python are listed below. They are organized into related groups based on the types of issues they deal with.

Language Exceptions

Description

StandardError

Base class for all built-in exceptions except StopIteration and SystemExit.

ImportError

Raised when an import statement fails.

SyntaxError

Raised when there is an error in Python syntax.

IndentationError

Raised when indentation is not specified properly.

NameError

Raised when an identifier is not found in the local or global namespace.

UnboundLocalError

Raised when trying to access a local variable in a function or method but no value has been assigned to it.

TypeError

Raised when an operation or function is attempted that is invalid for the specified data type.

LookupError

Base class for all lookup errors.

IndexError

Raised when an index is not found in a sequence.

KeyError

Raised when the specified key is not found in the dictionary.

ValueError

Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

RuntimeError

Raised when a generated error does not fall into any category.

MemoryError

Raised when a operation runs out of memory.

RecursionError

Raised when the maximum recursion depth has been exceeded.

SystemError

Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

Math Exceptions

Description

ArithmeticError

Base class for all errors that occur for numeric calculation. You know a math error occurred, but you don’t know the specific error.

OverflowError

Raised when a calculation exceeds maximum limit for a numeric type.

FloatingPointError

Raised when a floating point calculation fails.

ZeroDivisonError

Raised when division or modulo by zero takes place for all numeric types.

I/O Exceptions

Description

FileNotFoundError

Raised when a file or directory is requested but doesn’t exist.

IOError

Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. Also raised for operating system-related errors.

PermissionError

Raised when trying to run an operation without the adequate access rights.

EOFError

Raised when there is no input from either the raw\_input() or input() function and the end of file is reached.

KeyboardInterrupt

Raised when the user interrupts program execution, usually by pressing Ctrl+c.

Other Exceptions

Description

Exception

Base class for all exceptions. This catches most exception messages.

StopIteration

Raised when the next() method of an iterator does not point to any object.

AssertionError

Raised in case of failure of the Assert statement.

SystemExit

Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, it causes the interpreter to exit.

OSError

Raises for operating system related errors.

EnvironmentError

Base class for all exceptions that occur outside the Python environment.

AttributeError

Raised in case of failure of an attribute reference or assignment.

NotImplementedError

Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

All exceptions are objects. The classes that define the objects are organized in a hierarchy, which is shown below. This is important because the parent class of a set of related exceptions will catch all exception messages for itself and its child exceptions. For example, an ArithmeticError exception will catch itself and all FloatingPointError, OverflowError, and ZeroDivisionError exceptions.

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- StopAsyncIteration

+-- ArithmeticError

| +-- FloatingPointError

| +-- OverflowError

| +-- ZeroDivisionError

+-- AssertionError

+-- AttributeError

+-- BufferError

+-- EOFError

+-- ImportError

+-- LookupError

| +-- IndexError

| +-- KeyError

+-- MemoryError

+-- NameError

| +-- UnboundLocalError

+-- OSError

| +-- BlockingIOError

| +-- ChildProcessError

| +-- ConnectionError

| | +-- BrokenPipeError

| | +-- ConnectionAbortedError

| | +-- ConnectionRefusedError

| | +-- ConnectionResetError

| +-- FileExistsError

| +-- FileNotFoundError

| +-- InterruptedError

| +-- IsADirectoryError

| +-- NotADirectoryError

| +-- PermissionError

| +-- ProcessLookupError

| +-- TimeoutError

+-- ReferenceError

+-- RuntimeError

| +-- NotImplementedError

| +-- RecursionError

+-- SyntaxError

| +-- IndentationError

| +-- TabError

+-- SystemError

+-- TypeError

+-- ValueError

| +-- UnicodeError

| +-- UnicodeDecodeError

| +-- UnicodeEncodeError

| +-- UnicodeTranslateError

+-- Warning

+-- DeprecationWarning

+-- PendingDeprecationWarning

+-- RuntimeWarning

+-- SyntaxWarning

+-- UserWarning

+-- FutureWarning

+-- ImportWarning

+-- UnicodeWarning

+-- BytesWarning

+-- ResourceWarning

13.5. Principles for using Exceptions

There are many bad examples of exception use on the Internet. The purpose of an exception is to modify the flow-of-control, not to catch simple errors. If your try: except: block is in the same function that raises the exception, you are probably mis-using exceptions.

Principle 1:

If a condition can be handled using the normal flow-of-control, don’t use an exception!

Example 1:

DON’T DO THIS:

When you can just as easily test for no items in the list doing this:

try:

average = sum(a\_list) / len(a\_list)

except ZeroDivisionError:

average = 0

if len(a\_list) > 0:

average = sum(a\_list) / len(a\_list)

else:

average = 0

Example 2:

DON’T DO THIS:

When you can just as easily test for a valid index doing this:

try:

value = my\_list[index]

except IndexError:

value = -1

if 0 <= index < len(my\_list):

value = my\_list[index]

else:

value = -1

Example 3:

DON’T DO THIS:

When you can just as easily test to see if the key is valid doing this:

try:

value = my\_dictionary[key]

except KeyError:

value = -1

if key in my\_dictionary.keys():

value = my\_dictionary[key]

else:

value = -1

Principle 2:

If you call a function that potentially raises exceptions, and you can do something appropriate to deal with the exception, then surround the code that contains the function call with a try: except: block.

Example: Suppose you have a function that reads a file to set the state of an application when it starts up. You should catch any errors related to reading the file and set the state of the application to default values if they can’t be set from the file.

try:

load\_state('previous\_state.txt')

except OSError:

set\_state\_to\_defaults()

Principle 3:

If you call a function that potentially raises exceptions, and you can’t do anything meaningful about the conditions that are raised, then don’t catch the exception message(s).

13.6. Exceptions Syntax

There are many variations on the code that catches exceptions. Here is a brief summary, but other code variations are possible.

13.6.1. Catch All Exceptions

Catch all exceptions, regardless of their type. This will prevent your program from crashing, but this type of exception handling is rarely useful because you can’t do anything meaningful to recover from the abnormal condition.

try:

# Your normal code goes here.

# Your code should include function calls which might raise exceptions.

except:

# If any exception was raised, then execute this code block.

13.6.2. Catch A Specific Exception

This is perhaps the most often used syntax. It catches one specific condition and tries to re-cover from the condition.

try:

# Your normal code goes here.

# Your code should include function calls which might raise exceptions.

except ExceptionName:

# If ExceptionName was raised, then execute this block.

13.6.3. Catch Multiple Specific Exceptions

try:

# Your normal code goes here.

# Your code should include function calls which might raise exceptions.

except Exception\_one:

# If Exception\_one was raised, then execute this block.

except Exception\_two:

# If Exception\_two was raised, then execute this block.

else:

# If there was no exception then execute this block.

13.6.4. Clean-up After Exceptions

If you have code that you want to be executed even if exceptions occur, you can include a finally code block:

try:

# Your normal code goes here.

# Your code might include function calls which might raise exceptions.

# If an exception is raised, some of these statements might not be executed.

finally:

# This block of code will always execute, even if there are exceptions raised

13.6.5. An Example of File I/O

One place where you will always want to include exception handling is when you read or write to a file. Here is a typical example of file processing. Note that the outer try: except: block takes care of a missing file or the fact that the existing file can’t be opened for writing. The inner try: except: block protects against output errors, such as trying to write to a device that is full. The finally code guarantees that the file is closed properly, even if there are errors during writing.

try:

f = open("my\_file.txt", "w")

try:

f.write("Writing some data to the file")

finally:

f.close()

except IOError:

print "Error: my\_file.txt does not exist or it can't be opened for output."

13.7. Glossary

exception

An error that occurs at runtime.

handle an exception

To prevent an exception from terminating a program by wrapping the block of code in a try / except construct.

raise

To cause an exception by using the raise statement.