Exception Handling

chapter 6

In This Chapter

6.1 Introduction

When you create and develop programs, errors occur naturally. Sometimes, you misspell a name or keyword, or sometimes you unknowingly change the symbols. These are very common and easy to handle errors. But programming is not that easy and errors are not that simple. So, to handle virtually any type of errors that may occur, language developers have created numerous ways to catch and prevent them. Python also supports a specific and well-defined mechanism of catching and preventing errors of any type that may occur. This mechanism is known as **Exception Handling**. In this chapter you are going to learn about exception handling techniques in Python, different types of errors that may occur and ways to avoid them.

6.2 Exceptions and Exception Handling

Exception, in general, refers to some contradictory or unexpected situation or in short, an error that is unexpected. During program development, there may be some cases where the programmer does not have the certainty that this code-fragment is going to work right, either because it accesses to resources that do not exist or because it gets out of an unexpected range, etc. These types of anomalous situations are generally called exceptions and the way to handle them is called exception handling.

EXCEPTION

Contradictory or Unexpected situation or unexpected error during program execution, is known as **Exception**.

Broadly there are *two* types of errors :

(*i*) ***Compile-time errors.*** These are the errors resulting out of violation of programming language's grammar rules *e.g.*, writing syntactically incorrect statement like :

```
print ("A" + 2)
```

will result into compile-type error because of invalid syntax. All syntax errors are reported during compilation.

(*ii*) ***Run-time errors.*** The errors that occur during runtime because of unexpected situations. Such errors are handled through exception handling routines of Python. Exception handling is a transparent and nice way to handle program errors.

Many reasons support the use of exception handling. In other words, advantages of exception handling are :

(*i*) Exception handling separates error-handling code from normal code.

(*ii*) It clarifies the code (by removing error-handling code from main line of program) and enhances readability.

(*iii*) It stimulates consequences as the error-handling takes place at one place and in one manner.

(*iv*) It makes for clear, robust, fault-tolerant programs.

So we can summarize **Exception** as :

It is an exceptional event that occurs during runtime and causes normal program flow to be disrupted.

Some common examples of *Exceptions* are :

- Divide by zero errors
- Accessing the elements of an array beyond its range
- Invalid input
- Hard disk crash
- Opening a non-existent file
- Heap memory exhausted

EXCEPTION HANDLING

Way of handling anomalous situations in a program-run, is known as **Exception Handling**.

For instance, consider the following code :

```
>>> print (3/0)
```

If you execute above code, you'll receive an error message as :

```
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print 3/0
ZeroDivisionError: integer division or modulo by zero
```

This message is generated by default exception handler of Python. The default exception handler does the following upon occurrence of an Exception :

(*i*) Prints out exception description

(*ii*) Prints the ***stack trace***, *i.e.*, hierarchy of methods where the exception occurred

(*iii*) Causes the program to terminate.

6.3 Concept of Exception Handling

The global concept of error-handling is pretty simple. That is, write your code in such a way that it raises some error flag every time something goes wrong. Then trap this error flag and if this is spotted, call the error handling routine. The intended program flow should be somewhat like the one shown in Fig. 6.1.

The raising of imaginary error flag is called **throwing or raising an error**. When an error is thrown, the overall system responds by catching the error. And surrounding a block of error-sensitive-code-with-exception-handling is called trying to execute a block.

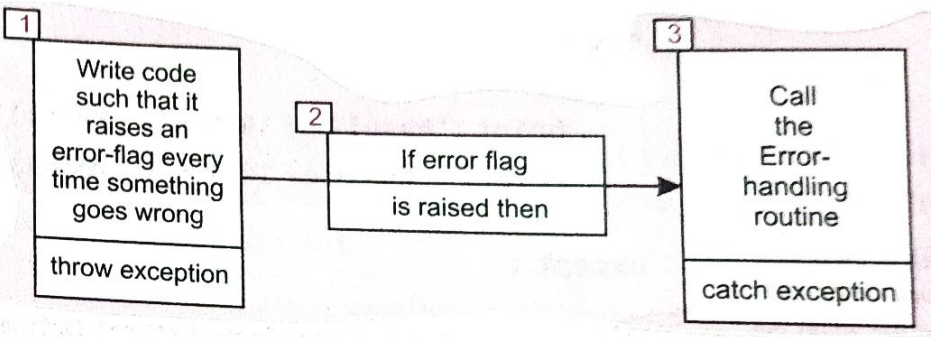

Figure 6.1 Concept of exception handling.

Some terminology used within exception handling follows.

Description	Python Terminology
An unexpected error that occurs during runtime	Exception
A set of code that might have an exception thrown in it.	try block
The process by which an exception is generated and passed to the program.	Throwing or raising an error
Capturing an exception that has just occurred and executing statements that try to resolve the problem	Catching
The block of code that attempts to deal with the exception (*i.e.*, problem).	*except clause or except/exception block or catch block*
The sequence of method calls that brought control to the point where the exception occurred.	Stack trace

When to Use Exception Handling

The exception handling is ideal for :

- processing exceptional situations.
- processing exceptions for components that cannot handle them directly.
- large projects that require uniform error-processing.

NOTE

An exception can be gener
by an error in your progran
explicitly *via* a **raise** stateme

NOTE

Unha

6.4 Exception Handling in Python

Exception Handling in Python involves the use of **try** and **except** clauses in the following format wherein the code that may generate an exception is written in the try block and the code for handling exception when the exception is raised, is written in except block.

See below :

```
try :
    #write here the code that may generate an exception
except :
    #write code here about what to do when the exception has occurred
```

For instance, consider the following code :

```
try :
    print ("result of 10/5 = ", (10/5))
    print ("result of 10/0 = ", (10/0))
except :
    print ("Divide by Zero Error! Denominator must not be zero!")
```

*The code that may raise an exception is written in **try block***

This is exception block : this will execute when the exception is raised

The output produced by above code is as shown below :

```
result of 10 / 5 =  2
result of 10 / 0 =  Divide by Zero Error! Denominator must not be zero!
```

*See, the expression (10 /0) raised exception which is then handled by **except block***

See, now the output produced does not show the scary red-coloured standard error message ; it is now showing what you defined under the exception block.

Consider another code that handles an exception raised when a code tries conversion from text to a number :

```
try:
    x = int("XII")
except:
    print ("Error converting 'XII' to a number")
```

The output generated from above code is not the usual error now, it is :

```
Error converting 'XII' to a number
```

Consider program 6.1 that is expanded version of the above example. It error- checks a user's input to make sure an integer is entered.

Program 6.1 Write a program to ensure that an integer is entered as input and in case any other value is entered, it displays a message – 'Not a valid integer'

```
ok = False
while not ok :
    try :
        numberString = input("Enter an integer:")
        n = int(numberString)
        ok = True
    except :
        print ("Error! Not a valid integer.")
```

```
Enter an integer: oo7
Error! Not a valid integer.
Enter an integer: 007
```

6.4.1 General Built-in Python Exceptions

In this section, we are discussing about some built-in exceptions of Python. The built-in exceptions can be generated by the interpreter or built-in functions. Some common built-in exceptions in Python are being listed below in Table 6.1.

Table 6.1 *Some built-in Exceptions*

Exception Name	Description
EOFError	Raised when one of the built-in functions (input( )) hits an end-of-file condition (EOF) without reading any data. (NOTE. the file.read( ) and file.readline( ) methods return an empty string when they hit EOF.)
IO Error	Raised when an I/O operation (such as a print statement, the built-in open( ) function or a method of a file object) fails for an I/O-related reason, *e.g.*, "file not found" or "disk full".
NameError	Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.
IndexError	Raised when a sequence subscript is out of range, *e.g.*, from a list of length 4 if you try to read a value of index like 8 or –8 etc. (Slice indices are silently truncated to fall in the allowed range ; if an index is not a plain integer, *TypeError* is raised.)
ImportError	Raised when an import statement fails to find the module definition or when a from ... import fails to find a name that is to be imported.
TypeError	Raised when an operation or function is applied to an object of inappropriate type, *e.g.*, if you try to compute a square-root of a string value. The associated value is a string giving details about the type mismatch.
ValueError	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as *IndexError*.
ZeroDivisionError	Raised when the second argument of a division or modulo operation is zero.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys
ImportError	Raised when the module given with import statement is not found.
KeyboardInterrupt	Raised when keys **Esc, Del** or **Ctrl+C** is pressed during program execution and normal program flow gets disturbed.

Following program 6.2 handles an error that may occur while opening a file.

Program 6.2 Program to handle exception while opening a file.

```
try:
    my_file = open("myfile.txt", "r")
    print (my_file.read())
except:
    print ("Error opening file")
```

```
Error opening file
```

The above program will open the file successfully if the file *myfile.txt* exists and contains some data otherwise it shows an output as :

Now the above output may be of one of the *two* reasons :

(*i*) the file did not exist or (*ii*) there was no data in the file.

But the above code did not tell which caused the error.

6.4.2 Second Argument of the except Block

You can also provide a second argument for the except block, which gives a reference to the exception object. You can do it in following format :

```
try:
    # code
except <ExceptionName> as <exArgument> :
    # handle error here
```

The *except clause* can then use this additional argument to print the associated error-message of this exception as : str (exArgument). Following code illustrates it :

```
try:
    print ("result of 10/5 = ", (10/5))
    print ("result of 10/0 = ", (10/0))
except ZeroDivisionError as e :
    print ("Exception - ", str(e))
```

Notice second argument to ***except*** *block i.e.,* ***e*** *here – gets reference of raised exception*

Printing standard error message of raised exception through the second argument

The above code will give output as :

```
result of 10/5 =  2.0
Exception - division by zero
```

The message associated with the exception

6.4.3 Handling Multiple Errors

Multiple types of errors may be captured and processed differently. It can be useful to provide a more exact error message to the user than a simple *"an error has occurred."* In order to capture and process different type of exceptions, there must be multiple exception blocks – each one pertaining to different type of exception.

This is done as per following format :

```
try:
    # :
except <exceptionName1> :
    # :
except <exceptionName2> :
    # :
except :
    # :
else :
    #If there is no exception then the statements in this block get executed.
```

*The **except** block without any exception name will handle the rest of the exceptions*

The last *else* : clause will execute if there is no exception raised, so you may put your code that you want to execute when no exceptions get raised. Following program 6.3 illustrates the same.

Program 6.3 Program to handle multiple exceptions.

```
try:
    my_file = open("myfile.txt")
    my_line = my_file.readline()
    my_int = int(s.strip())
    my_calculated_value = 101 / my_int

except IOError:
    print ("I/O error occurred")

except ValueError:
    print ("Could not convert data to an integer.")

except ZeroDivisionError:
    print ("Division by zero error")

except:
    print ("Unexpected error:")
else :
    print ("Hurray! No exceptions!")
```

*These three except blocks will catch and handle **IOError** , **ValueError** and **ZeroDivisionError** exceptions respectively*

*The unnamed **except** block will handle the rest of the exceptions*

This last else: block will get executed if no exception is raised

The output produced by above code is :

```
I/O error occurred
```

Exception Handling – execution order

The <try suite> is executed first ;

if, during the course of executing the <try suite>, an exception is raised that is not handled otherwise, and

the <except suite> is executed, with <name> bound to the exception, if found ; if no matching except suite is found then *unnamed except suite* is executed.

NOTE

The **named except:** blocks handle the named exceptions while the **unnamed except:** block handles all other exceptions – exceptions not mentioned in named except: blocks.

6.4.4 The finally Block

You can also use a **finally**: block along with a **try**: block, just like you use **except**: block, e.g., as:

```
try:
    # statements that may raise exception
[except:
    # handle exception here]
finally:
    # statements that will always run
```

The difference between an **except:** block and the **finally:** block is that the **finally:** block is a place that contains any code that must execute, whether the **try:** block raised an exception or not. For example,

```
try:
    fh = open("poems.txt", "r+")
    fh.write("Adding new line")
finally:
    print ("Error: can\'t find file or read data")
```

This statement will always be executed in the end

You may combine **finally:** with **except:** clause. In such a combination, the **except:** block will get executed only in case an exception is raised and **finally:** block will get executed ALWAYS, in the end. Following code illustrates it :

```
try:
    fh = open("poem1.txt", "r")
    print (fh.read())
except:
    print ("Exception Occurred")
finally:
    print ("Finally saying goodbye.")
```

The output produced by above code will be :

```
Exception Occurred
Finally saying goodbye
```

*This is printed because **finally:** block got executed in the end.*

*This is printed because **except:** block got executed when exception occurred.*

In the above code if no exception is raised, still the above code will print :

```
Finally saying goodbye
```

because **finally :** block gets executed always in the end.

6.4.5 Raising/Forcing an Exception

In Python, you can use the **raise** keyword to raise/force an exception. That means, you as programmer can force an exception to occur through **raise** keyword. It can also pass a custom message to your exception handling module. For example :

```
raise <exception> (<message>)
```

The exception raised in this way should be a pre-defined Built-in exception. Consider following code snippet that forces a *ZeroDivisionError* exception to occur without actually dividing a value by zero :

```
try :
    a = int(input("Enter numerator :"))
    b = int(input("Enter denominator :"))
    if b == 0 :
        raise ZeroDivisionError(str(a) + "/0 not possible")
    print (a/b)
except ZeroDivisionError as e :
    print ("Exception", str(e))
```

*Notice this **raise** statement is raising built-in exception ZeroDivisionError with a custom message, that follows the exception name*

The output produced by above code is :

```
Enter numerator : 7
Enter denominator : 0
Exception 7/0 not possible
```

*This was the custom-message sent; printed as **str(e)** because the exception reference was received in **e***

Python assert Statement

In some situations, you have a clear idea about the requirements and test-conditions required. So in programs where you know the likely results of some conditions and where results being different from the expected results can crash the programs, you can use Python **assert** statement if the condition is resulting as expected or not.

Python's **assert** statement is a debugging aid that tests a condition. If the condition is *true*, it does nothing and your program just continues to execute. But if the assert condition evaluates to *false*, it raises an **AssertionError** *exception* with an optional error message. The syntax of **assert** statement in Python, is :

```
assert condition [, error_message ]
```

Specify the optional, custom message through error_message

For example, consider the following code :

```
print("Enter the Numerator: ")
n = int(input())
print("Enter the Denominator: ")
d = int(input())
assert d != 0, "Denominator must not be 0"
print("n/d =", int(n/d))
```

Custom Error message specified

*This error message will be printed only when the given condition **d != 0** results into **false**.*

NOTE

The **assert** keyword in Python is used when we need to detect problems early.

Check Point 6.1

1. Name the block that encloses the code that may encounter anomalous situations.
2. In which block can you raise the exception ?
3. Which block traps and handles an exception ?
4. Can one except block sufficiently trap and handle multiple exceptions ?
5. Which block is always executed no matter which exception is raised ?
6. Name the root class for all exceptions.

Benefits of Exception Handling

Exception provides the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In short, the advantages of exception handling are :

(*i*) Exception handling separates error-handling code from normal code.

(*ii*) It clarifies the code and enhances readability.

(*iii*) It stimulates consequences as the error-handling takes place at one place and in one manner.

(*iv*) It makes for clear, robust, fault-tolerant programs.

NOTE

All exceptions are sub-classes of *Exception* class.