Exception Handling in Python

Sometimes, we want to catch some or all of the errors that could get generated during program execution and as a programmer, we need to be as specific as possible. Therefore, Python allows programmers to deal with errors efficiently.

Exceptions are events that are used to modify the flow of control through a program when the error occurs. Exceptions get triggered automatically on finding errors in Python.

These exceptions are processed using five statements. These are:

1. try/except: catch the error and recover from exceptions hoist by programmers or Python itself.
2. try/finally: Whether exception occurs or not, it automatically performs the clean-up action.
3. assert: triggers an exception conditionally in the code.
4. raise: manually triggers an exception in the code.
5. with/as: implement context managers in older versions of Python such as - Python 2.6 & Python 3.0.
6. The last was an optional extension to Python 2.6 & Python 3.0.

Why are Exceptions Used?

Exceptions allow us to jump out of random, illogical large chunks of codes in case of errors. Let us take a scenario that you have given a function to do a specific task. If you went there and found those things missing that are required to do that particular task, what will you do? Either you stop working or think about a solution - where to find those items to perform the task. The same thing happens here in case of Exceptions also. Exceptions allow programmers to jump an exception handler in a single step, abandoning all function calls. You can think exceptions to an optimized quality go-to statement, in which the program error that occurs at runtime gets easily managed by the exception block. When the interpreter encounters an error, it lets the execution go to the exception part to solve and continue the execution instead of stopping.

While dealing with exceptions, the exception handler creates a mark & executes some code. Somewhere further within that program, the exception is raised that solves the

problem & makes Python jump back to the marked location; by not throwing away/skipping any active functions that were called after the marker was left.

Roles of an Exception Handler in Python

- Error handling: The exceptions get raised whenever Python detects an error in a program at runtime. As a programmer, if you don't want the default behavior, then code a 'try' statement to catch and recover the program from an exception. Python will jump to the 'try' handler when the program detects an error; the execution will be resumed.
- Event Notification: Exceptions are also used to signal suitable conditions & then passing result flags around a program and text them explicitly.
- Terminate Execution: There may arise some problems or errors in programs that it needs a termination. So try/finally is used that guarantees that closing-time operation will be performed. The 'with' statement offers an alternative for objects that support it.
- Exotic flow of Control: Programmers can also use exceptions as a basis for implementing unusual control flow. Since there is no 'go to' statement in Python so that exceptions can help in this respect.

A Simple Program to Demonstrate Python Exception Handling:

Example 01:

```
(a,b) = (6,0)
try:# simple use of try-except block for handling errors
   g = a/b
except ZeroDivisionError:
    print ("This is a DIVIDED BY ZERO error")
```

Output:

This is a DIVIDED BY ZERO error

The above program can also be written like this:

Example 02:

```
(a,b) = (6,0)
try:
```

```
    g = a/b
except ZeroDivisionError as s:
    k = s
    print (k)
#Output will be: integer division or modulo by zero
```

Output:

```
division by zero
```

The 'try - Except' Clause with No Exception

The structure of such type of 'except' clause having no exception is shown with an example.

```
try:
        # all operations are done within this block.
        . . . . . .
except:
        # this block will get executed if any exception
encounters.
        . . . . . .
else:
        # this block will get executed if no exception is found.
        . . . . . .
```

All the exceptions get caught where there is try/except the statement of this type. Good programmers use this technique of exception to make the program fully executable.

'Except' clause with multiple exceptions:

```
try:
        # all operations are done within this block.
        . . . . . .
except ( Exception1 [, Exception2[,….Exception N ] ] ] ) :
```

```
        # this block will get executed if any exception
encounters from the above lists of exceptions.

        . . . . . .

else:

        # this block will get executed if no exception is found.

      . . . . . .
```

The 'try-finally' clause

```
try:

        # all operations are done within this block.

        . . . . . .

        # if any exception encounters, this block may get
skipped.

finally:

         . . . . . .

        # this block will definitely be executed.
```

Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities in them **–**

- **Exception Handling –** This would be covered in this tutorial. Here is a list standard Exceptions available in Python:
- This would be covered in Assertions in Python

List of Standard Exceptions **–**

Sr.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.

3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisionError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError**

	Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.
14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**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.
21	**IOError**

	Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.
28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

Assertions in Python

An assertion is a sanity-check that you can turn on or turn off when you are done with your testing of the program.

The easiest way to think of an assertion is to liken it to a **raise-if** statement (or to be more accurate, a raise-if-not statement). An expression is tested, and if the result comes up false, an exception is raised.

Assertions are carried out by the assert statement, the newest keyword to Python, introduced in version 1.5.

Programmers often place assertions at the start of a function to check for valid input, and after a function call to check for valid output.

The *assert* Statement

When it encounters an assert statement, Python evaluates the accompanying expression, which is hopefully true. If the expression is false, Python raises an *AssertionError* exception.

The **syntax** for assert is –

```
assert Expression[, Arguments]
```

If the assertion fails, Python uses ArgumentExpression as the argument for the AssertionError. AssertionError exceptions can be caught and handled like any other exception using the try-except statement, but if not handled, they will terminate the program and produce a traceback.

Example

Here is a function that converts a temperature from degrees Kelvin to degrees Fahrenheit. Since zero degrees Kelvin is as cold as it gets, the function bails out if it sees a negative temperature –

```
def KelvinToFahrenheit(Temperature):
   assert (Temperature >= 0),"Colder than absolute zero!"
   return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(273)
print int(KelvinToFahrenheit(505.78))
print KelvinToFahrenheit(-5)
```

When the above code is executed, it produces the following result –

```
32.0
451
Traceback (most recent call last):
File "test.py", line 9, in <module>
```

```
print KelvinToFahrenheit(-5)
File "test.py", line 4, in KelvinToFahrenheit
assert (Temperature >= 0),"Colder than absolute zero!"
AssertionError: Colder than absolute zero!
```

What is Exception?

An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a **try:** block. After the try: block, include an **except:** statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks –

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax –

- A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
- You can also provide a generic except clause, which handles any exception.
- After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.
- The else-block is a good place for code that does not need the try: block's protection.

Example

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

This produces the following result −

```
Written content in the file successfully
```

Example

This example tries to open a file where you do not have write permission, so it raises an exception −

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This produces the following result −

```
Error: can't find file or read data
```

The *except* Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows −

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a **try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because

it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* Clause with Multiple Exceptions

You can also use the same *except* statement to handle multiple exceptions as follows –

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

You can use a **finally:** block along with a **try:** block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this –

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

You cannot use *else* clause as well along with a finally clause.

Example

```
try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
finally:
   print "Error: can\'t find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result –

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows –

```
try:
```

```
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print "Going to close the file"
      fh.close()
except IOError:
   print "Error: can\'t find file or read data"
```

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
try:
   You do your operations here;
   ......................
except ExceptionType, Argument:
   You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

Example

Following is an example for a single exception –

```
# Define a function here.
def temp_convert(var):
   try:
      return int(var)
   except ValueError, Argument:
      print "The argument does not contain numbers\n", Argument

# Call above function here.
```

```
temp_convert("xyz");
```

This produces the following result –

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the **raise** statement is as follows.

Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (for example, NameError) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows –

```
try:
   Business Logic here...
except "Invalid level!":
   Exception handling here...
else:
   Rest of the code here...
```

User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable e is used to create an instance of the class *Networkerror*.

```
class Networkerror(RuntimeError):
   def __init__(self, arg):
      self.args = arg
```

So once you defined above class, you can raise the exception as follows −

```
try:
   raise Networkerror("Bad hostname")
except Networkerror,e:
   print e.args
```