

Python Exception Handling

Session 7

Objectives

- ▶ Exception Handling
- ▶ User-defined Exception

Python - Exceptions Handling

- ▶ 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: Standard Exceptions.
 - ▶ **Assertions** – This would be covered in Assertions in Python tutorial.
- ▶ 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.

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.

What is Exception ?

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.

Handling Exception

```
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 –

```
#!/usr/bin/python

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"
```

[Live Demo](#)

This produces the following result –

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

Handling Exception

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.
```

Handling Exception

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.

Handling Exception

```
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 –

```
#!/usr/bin/python

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"
```

[Live Demo](#)

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.

Argument of an 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

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
```


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.

Assertions in Python

```
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!
```

Summary

- ▶ Exception Handling
- ▶ User-defined Exception