Python Exceptions Handling

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

- **Exception Handling:**
- **Assertions:**

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 can't 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 would terminate and come out.

Handling an exception:

- If there is a *suspicious* code that may raise an exception, we can defend 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:

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

Important points

- 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:

```
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 will produce following result:

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

The *except* clause with no exceptions:

```
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:

```
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.

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

Note that you can provide except clause(s), or a finally clause, but not both.

You can not 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 following result:

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

Reading and Writing Files:

The *file* object provides a set of access methods to make our lives easier. We would see how to use *read()* and *write()* methods to read and write files.

The *write()* Method:

- The *write()* method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- The write() method does not add a newline character ('\n') to the end of the string:

Syntax:

```
fileObject.write(string);
```

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 are writing 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 will receive 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:

```
def temp_convert(var):
  try:
        return int(var)
  except ValueError, Argument:
        print "The argument does not
  contain numbers\n", Argument
temp_convert("xyz");
```

- This would produce 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.

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:

```
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 our except clause as follows:

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

- 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
- 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
- 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
- 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
- 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.

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 your exception as follows:

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

```
• try:
•     a = int(input("Enter a:"))
•     b = int(input("Enter b:"))
•     c = a/b
• except:
•     print("Can't divide with zero")
```

The except statement with no exception

- **try**:
-     a = int(input("Enter a:"))
-     b = int(input("Enter b:"))
-     c = a/b
-     **print**("a/b = %d"%c)
- # Using Exception with except statement. If we pri nt(Exception) it will return exception class
- **except** Exception:
-     **print**("can't divide by zero")
-     **print**(Exception)
- **else**:
-     **print**("Hi I am else block")

The except statement using with exception variable

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Points to remember

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.