TATA ELXSI

Python Scripting

Learning & Development Team

Monday, November 2, 2020

Module – 5 : Errors and Exceptions

Agenda

- Errors
- Try..Except
- Handling Exceptions
- Raising Exceptions
- How To Raise Exceptions
- Try..Finally
- Using Finally

Errors and Exceptions

- Syntax Errors : Syntax errors, also known as parsing errors

```
>>> while True
print('Hello world')

File "<stdin>", line 1, in ?

while True print('Hello world')

                ^

SyntaxError: invalid syntax
```

Exceptions

Errors detected during execution are called exceptions

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops!  That was no valid number.  Try again...")
```

Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: int division or modulo by zero

>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

Handling Exceptions

- It is possible to write programs that handle selected exceptions.

```
>>> while True :
...   try :
...         x =int(input("Enter a number"))
...         break
...   except ValueError :
...         print ("OOPS error")

Enter a number
OOPS error
Enter a number
OOPS error
```

Try statement

- The try statement works as follows.
  - First, the try clause (the statement(s) between the try and ‘except’ keywords) is executed.
  - If no exception occurs, the except clause is skipped and execution of the try statement is finished.
  - If an exception occurs during execution of the try clause, the rest of the clause is skipped.
    - Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
  - If an exception occurs which does not match the exception named in the except clause, it is passed on to outer ‘try’ statements;
  - if no handler is found, it is an unhandled exception and execution stops

Try statement

- A try statement may have more than one except clause, to specify handlers for different exceptions.
- Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement.
- An except clause may name multiple exceptions as a parenthesized tuple, for example:
- ... except (RuntimeError, TypeError, NameError):
- ... pass

Example

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
I/O error: [Errno 2] No such file or directory: 'myfile.txt'
>>> ================================
Could not convert data to an integer.
```

else clause - Example

- The try ... except statement has an optional else clause, which, when present, must follow all except clauses.

- It is useful for code that must be executed if the try clause does not raise an exception.

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Try - except

- The use of the ‘else’ clause is better than adding additional code to the ‘try’ clause.

- it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try ... except statement.

- Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause.

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
```

- Handling run-time error: int division or modulo by zero

Raising Exceptions

- The raise statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

- The sole argument to ‘raise’ indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from ‘Exception’).

Re-Raising Exceptions

- If you need to determine whether an exception was raised but don't intend to handle it,

- A simpler form of the raise statement allows you to re-raise the exception:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

User-defined Exceptions

- Programs may name their own exceptions by creating a new exception class.
- Exceptions should typically be derived from the Exception class, either directly or indirectly.

User-defined Exceptions

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception occurred, value:', e.value)
...finally:
…     print(“I will be executed always”)
```

```
My exception occurred, value: 4
I will be executed always
```

Thank you

Learning & Development

Tata Elxsi

Bangalore

www.tataelxsi.com