**Unit-VIII**

**Exception Handling**

**Errors in program:-**

Errors or mistakes in a program are often referred to as bugs. They are almost always the fault of the programmer. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

* Syntax errors
* Runtime errors
* Logical errors

**Syntax errors**

Syntax errors are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors. Some syntax errors can be caught and handled, like eval(""), but these are rare.In IDLE, it will highlight where the syntax error is. Most syntax errors are typos, incorrect indentation, or incorrect arguments. If you get this error, try looking at your code for any of these.

Common Python syntax errors include:

* leaving out a keyword
* putting a keyword in the wrong place
* leaving out a symbol, such as a colon, comma or brackets
* misspelling a keyword
* incorrect indentation
* empty block

**Logic errors**

These are the most difficult type of error to find, because they will give unpredictable results and may crash your program. A lot of different things can happen if you have a logic error. However these are very easy to fix as you can use a debugger, which will run through the program and fix any problems.

Here are some examples of mistakes which lead to logical errors:

* using the wrong variable name
* indenting a block to the wrong level
* using integer division instead of floating-point division
* getting operator precedence wrong
* making a mistake in a boolean expression
* off-by-one, and other numerical errors

**Runtime errors**

If a program is syntactically correct – that is, free of syntax errors – it will be run by the Python interpreter. However, the program may exit unexpectedly during execution if it encounters a *runtime error* – a problem which was not detected when the program was parsed, but is only revealed when a particular line is executed. When a program comes to a halt because of a runtime error, we say that it has crashed.

Consider the English instruction *flap your arms and fly to Australia.* While the instruction is structurally correct and you can understand its meaning perfectly, it is impossible for you to follow it.

Some examples of Python runtime errors:

* division by zero
* performing an operation on incompatible types
* using an identifier which has not been defined
* accessing a list element, dictionary value or object attribute which doesn’t exist
* trying to access a file which doesn’t exist

**Exceptions**

Exceptions arise when the python parser knows what to do with a piece of code but is unable to perform the action. An example would be trying to access the internet with python without an internet connection; the python interpreter knows **what to do with that command but is unable to perform it.**

## 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**

***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.

# Python program to handle simple runtime error

 a = [1, 2, 3]

try:

    print "Second element = %d" %(a[1])

    print "Fourth element = %d" %(a[3])

except IndexError:

    print "An error occurred"

**Output:**

Second element = 2

An error occurred

**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()

**Output:-**

Written content in the file successfully

# Program to handle multiple errors with one except statement

try :

a = 3

if a < 4 :

b = a/(a-3)

print "Value of b = ", b

except(ZeroDivisionError, NameError):

print "\nError Occurred and Handled"

**Example:-**

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"

**Output:-**

Error: can't find file or read data

## 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

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.

**Example:-**

try:

a=10/0;

except(ArithmeticError, IOError):

print("Arithmetic Exception")

else:

print("Successfully Done")

## 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.

**Syntax:-**

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"

**Output-**

Error: can't find file or read data

**Example:-**

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

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

def temp\_convert(var):

try:

return int(var)

except ValueError, Argument:

print "The argument does not contain numbers\n", Argument

temp\_convert("xyz");

**Output:-**

The argument does not contain numbers

invalid literal for int() with base 10: 'xyz'

**Raising Exception:**  
 The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

|  |  |  |
| --- | --- | --- |
| **Example:-**   try:      raise NameError("Hi there")  except NameError:      print "An exception"    **Types of Exception:-** | | |
| **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 | | **OSError**  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. |

#### OverFlow Error :-

The Overflow Error is raised when the result of an arithmetic operation is out of range. OverflowError is raised for integers that are outside a required range.

## Example:-

## try:

## import math

## print(math.exp(1000))

## except OverflowError:

## print ("OverFlow Exception Raised.")

## else:

## print ("Success, no error!")

### Assertion Error :-

When an assert statement is failed, an Assertion Error is raised. Let's take an example to understand the assertion error. Let's say you have two variables a and b, which you need to compare. To check whether a and b are equal or not, you apply an assert keyword before that, which will raise an Assertion exception when the expression will return false.

**Example:-**

try:

a = 100

b = "DataCamp"

assert a == b

except AssertionError:

print ("Assertion Exception Raised.")

else:

print ("Success, no error!")

Assertion Exception Raised.

## Lookup Error:-

## Lookup Error acts as a base class for the exceptions that occur when a key or index used on a mapping or sequence of a list/dictionary is invalid or does not exists.

## The two types of exceptions raised are:

## 1) IndexError

## 2) KeyError

## Key Error:-

## If a key you are trying to access is not found in the dictionary, a key error exception is raised.

## Example:-

## try:

## a = {1:'a', 2:'b', 3:'c'}

## print (a[4])

## except LookupError:

## print ("Key Error Exception Raised.")

## else:

## print ("Success, no error!")

## Index Error:-

## When you are trying to access an index (sequence) of a list that does not exist in that list or is out of range of that list, an index error is raised.

## Example:-

## try:

## a = ['a', 'b', 'c']

## print (a[4])

## except LookupError:

## print ("Index Error Exception Raised, list index out of range")

## else:

## print ("Success, no error!")

## Memory Error

## As discussed earlier, Memory Error is raised when an operation does not get enough memory to process further.

## Name Error

## Name Error is raised when a local or global name is not found.

## In the below example, ans variable is not defined. Hence, you will get a name error.

## Example:-

## try:

## print (ans)

## except NameError:

## print ("NameError: name 'ans' is not defined")

## else:

## print ("Success, no error!")

## Output:-

## NameError: name 'ans' is not defined

## Type Error:-

## Type Error Exception is raised when two different or unrelated types of operands or objects are combined.

## In the below example, an integer and a string are added, which results in a type error.

## Example:-

## try:

## a = 5

## b = "DataCamp"

## c = a + b

## except TypeError:

## print ('TypeError Exception Raised')

## else:

## print ('Success, no error!')

## Value Error :-

## Value error is raised when the built-in operation or a function receives an argument that has a correct type but invalid value.

## In the below example, the built-in operation float receives an argument, which is a sequence of characters (value), which is invalid for a type float.

## Example:-

## try:

## print (float('DataCamp'))

## except ValueError:

## print ('ValueError: could not convert string to float: \'DataCamp\'')

## else:

## print ('Success, no error!')

## 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.

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

**Example:-**

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

**Example:-**

class MyError(Exception):

     def \_\_init\_\_(self, value):

        self.value = value

     def \_\_str\_\_(self):

        return(repr(self.value))

try:

    raise(MyError(3\*2))

except MyError as error:

    print('A New Exception occured: ',error.value)

**Example:-**

try:

    age = int(input("Enter the age:"))

    if(age<18):

        raise ValueError

    else:

        print("the age is valid")

except ValueError:

    print("The age is not valid")