Scripting Language

UNIT-III

K.RaviKanth, Ast.Prof., Dept. of CSE, IIIT- RGUKT- Basara, T.S, IND.

SYLLABUS

Errors and Exception Handling:

- Introduction to Errors and Exceptions
- Types of Errors
- Types of Exceptions
- Multiple Except Blocks
- Raising Exceptions
- Built-in and User-defined Exceptions

Introduction to Errors and Exceptions

- The programs that we write may behave abnormally or unexpectedly because of some errors and/or exceptions. The two common types of errors that we very often encounter are *syntax errors* and *logic errors*. While logic errors occur due to poor understanding of problem and its solution, syntax errors, on the other hand, arises due to poor understanding of the language. However, such errors can be detected by exhaustive debugging and testing of procedures.
- Exceptions are run-time anomalies or unusual conditions (such as divide by zero, accessing arrays out of its bounds, running out of memory or disk space, overflow, and underflow) that a program may encounter during execution.
- Like errors, exceptions can be categorized as synchronous and asynchronous exceptions. Synchronous exceptions (like divide by zero, array index out of bound, etc.) can be controlled by the program, asynchronous exceptions (like an interrupt from the keyboard, hardware malfunction, or disk failure), on the other hand, are caused by events that are beyond the control of the program.

Types of Errors

Syntax Errors:

•occurs when we violate the rules of Python and they are the most common kind of error that we get while learning a new language. For example, consider the lines of code given below.

•These are syntactical errors found in the code, due to which a program fails to execute.

•For example, forgetting a colon at the end of the a python function program, or writing a statement without proper syntax will result in syntax errors or parsing error.

```
i=0
if i == 0 print(i)

SyntaxError: invalid syntax
```

Logical Errors

Ex: Problem with the logic in the program

- specifies all those type of errors in which the program executes but gives incorrect results.
- Logical error may occur due to wrong algorithm or logic to solve a particular program.
- In some cases, logic errors may lead to divide by zero or accessing an item in a list where the index of the item is outside the bounds of the list.
- In this case, the logic error leads to a run-time error that causes the program to terminate abruptly.
- These types of run-time errors are known as *exceptions*.
- Logical errors are not detected either by the python interpreter

Ex: EH_Logical.py

```
def sum(x,y):
    c=x-y
    return c
print("sum is",sum(50,30))
```

Output: sum is 20

```
>>> 10/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    10/0
ZeroDivisionError: division by zero
>>> num+20
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    num+20
NameError: name 'num' is not defined
>>> 'Roll No'+ 123
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    'Roll No'+ 123
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Run time errors

These are the errors which represent inefficiency of the computer system to execute a particular statement, means computer system can not process.

For example, division by zero error occur at run-time.

Ex: Before exception Handling:

Division by Zero Error

```
(x,y)=(5,0)
z=x/y
print(z)
print("Iam sorry")
```

```
Output: Traceback (most recent call last):
        File "C:/Users/RAVI@RGUKT/Desktop/EH2.py", line 2, in <module>
        z=x/y
        ZeroDivisionError: division by zero
```

NOTE: When we run the program, the python interpreter encounters an error of dividing an integer by zero & therefore creates an exception object & throws it i.e, informs us that an error has occurred

Therefore it produces a message

"ZeroDivisionError: division by zero

" & the program terminates without executing the remaining statements

If we want the pgm to continue with the execution of the remaining code, then we should handle the execution properly, which is called **"Exception Handling"**

After exception Handling:

```
(x,y) = (5,0)
try:
    z = x/y
except ZeroDivisionError as e:
    z=e
    print(z)
    print("hello")
```

Output: division by zero
hello

What is an Exception?

- An Exception is an abnormal situation (or) unexpected situation in the normal flow of the program execution.
- An exception is an abnormal event that rises during the execution of a program & disturbs the normal flow of instructions
- In other words, an exception is a run-time error. Therefore, we can say that an exception is a Python object that represents an error.
- Because of Exceptions the flow of program execution is getting disturbed so that program execution may continue (or) may not be continued.
  - Examples
    - Division by zero
    - Attempts to use an array index out of bounds.
    - Number input in wrong format
    - Unable to write output to file
    - Missing input file

Exception Handling

- Whenever an exception is occurred, handling those exceptions called as "Exception Handling".
- Performing action in response to exception
- Examples
  - Exit program (abort)
  - Ignore exception
  - Deal with exception and continue

Print error message Request new data Retry action

Benefits of Exception Handling

- It allows us to fix the error.
- It prevents program from automatically terminating.
- Separates Error-Handling Code from Regular Code.
  - Conventional programming combines error detection, reporting, handling code with the regular code, leads to confusion.

Exceptional Handling involves the following:

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take Corrective actions (Handle the exception)

When writing program, we must always be on the lookout for places in the program where an exception could be generated

Exceptions are of various types & all exception types are subclasses of built-in class "**Exception**"

This class is available in "**import sys**" module, which contains all the errors & exceptions

Exception

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- An exception is a Python object that represents an error.
- Used to handle any unexpected error in python programs

- **Examples:**
  - When a file we try to open does not exist (FileNotFoundError)
  - dividing a number by zero (ZeroDivisionError)
  - module we try to import is not found (ImportError) etc

- Whenever above type of runtime error occur, Python creates an exception object.
- If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Few Standard Exceptions/Python Built-in Exceptions

Exception Name	Description
Exception	Base class for all exceptions
Arithmetic Error	Base class for all errors that are generated due to mathematical calculations
Floating Point Error	Raised when a floating point calculation could not be performed
Zero Division Error	Raised when division or modulo by zero takes place for all numeric types
IO Error	Raised when an input/output operation fails such as print( ) or open( ) functions when trying to open a file that does not exist
Syntax Error	Raised when there is a error on Python code
Indentation Error	Raised when indentation problem in pgm

Exception Name	Description
Value Error	Raised when built-in functions for a data type has a valid type of arguments, but the arguments have invalid values specified
Runtime Error	Raised when a generated error does not fall into any category
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when end of file is reached or there is no input for input() function
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.

Exception Name	Description
IndexError	Raised when index is not found in a sequence
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Generated when the next( ) method of an iterator does not point to any object

Exception Name	Description
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
Standard Error	Base class for all built-in exceptions (excluding StopIteration and SystemExit)
Lookup Error	Base class for all lookup errors

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, includes an except: statement, followed by a block of code which handles the problem as elegantly as possible
- Different ways of Exception Handling in python are:
- try …except…else
- try… except
- try ... Finally
- Note: The try ... except block can optionally have an else clause, which, when present, must follow all except blocks. The statement(s) in the else block is executed only if the try clause does not raise an exception.

- In Python, Exception Handling is managed through three (03) keywords : **try, except, finally**

try Block:

The statements that are expected to produce exception are identified in the pgm & are placed with in a "try block"

Syntax: try:

//Body of code to monitor for errors

Note: If an exception occurs within the try block, the appropriate exception handler (catch block) associated with the try block handles the exception immediately

except Block:

The "except block" is used to process the exception raised. The except block is placed immediately after the try block.

Syntax: except Exception_Type Exception_Object:

// exception handler code for the Exception Type

No Statements are placed btw try block & except block. Python also supports multiple except stmts. For a single try block,

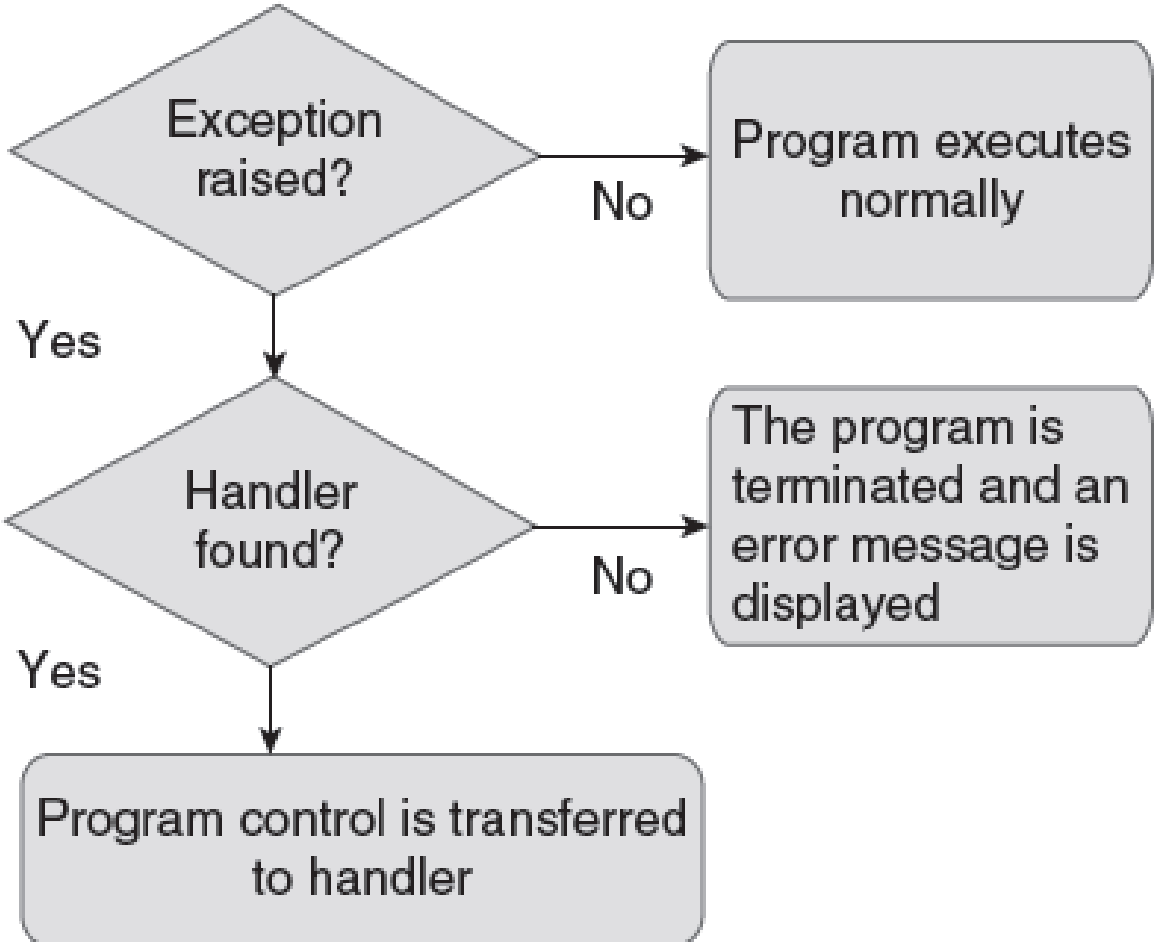
This is useful when more than one exception could be raised by a single piece of code.

In such situations specify two or more except blocks, each specifying diff. types of exception

A subclass must come before their super classes in a series of catch sts. " Exception" class is the super class of all the exceptions

Handling Exceptions

```
try:
    statements
except ExceptionName:
    statements
```



```
num=int(input("Enter the Numerator:"))
deno=int(input("Enter the Denominator:"))
try:
    quo=num/deno
    print("Quotient:",quo)
except ZeroDivisionError:
    print("Denominator cannot be zero")
```

```
>>>
Enter the Numerator:20
Enter the Denominator:10
Quotient: 2.0
>>> ================================
>>>
Enter the Numerator:20
Enter the Denominator:0
Denominator cannot be zero
>>>
```

```
try:

    -------
    -------

except ArithmeticException ae:

    ----------
    ----------

except Exception e:

    ----------
    ----------
else:
        -------
```

The control enters into the try block & executes the statements

If an exception is raised the exception is caught by the particular except block & the stmts in that except block are executed

The control never comes back again to the try block

If no exception are raised, except blocks are not executed & the control directly passes to the next stmts, after all the except blocks

finally block:

The “finally block” can be placed immediately after the try block or after the last except block, if except blocks exists.

Syntax:

finally :

// block of code to be executed before the exception handling mechanism ends

Ex: try:

----

finally:

(or)

Ex: try:

----

except -- --:

----

except:

……..

……..

finally:

K.RaviKanth, Ast.Prof., Dept. of CSE, IIIT- RGUKT-Basara, T.S, IND.

The finally block is executed regardless of whether or not an exception is raised

Therefore we can use it to perform certain house-keeping operations like closing files, releasing system resources, destroying objects, deallocating memory

It can also be used to handle an exception that is not caught by any of the previous except statements

When a pgm comprises of a try block, except block(s), finally block & if no exceptions are raised then except blocks are not executed, the control directly passed to the finally block after try block

```
num=int(input("Enter the Numerator:"))
deno=int(input("Enter the Denominator:"))
try:
    quo=num/deno
    print("Quotient:",quo)
except ZeroDivisionError:
    print("Denominator cannot be zero")
finally:
    print("Job Done")
```

```
Enter the Numerator:30
Enter the Denominator:30
Quotient: 1.0
Job Done

>>> ================================ RES
>>>
Enter the Numerator:30
Enter the Denominator:0
Denominator cannot be zero
Job Done
```

Handling an Exception try…except…else

- A single try statement can have multiple except statemnts
- Useful when we have a try block that may throw different types of exception
- Code in else-block executes if the code in the try:block does not raise an exception
- **Syntax:**

```
try:
        you do your operations here
        ……….……….
except ExceptionA:
        If there is ExceptionA, then execute this block
 except ExceptionB:
        If there is ExceptionB, then execute this block
……..……..
else:
        If there is no exception then execute this block
```

Ex:

```
try:
    fh=open("testfile.txt","w")
    fh.write("This is my test file")
except IOError:
    print("Error: can't find or read data")
else:
    print("Written content to file successfully")
    fh.close()
```

Output: Written content to file successfully

Note: Here you try to open the same file when you do not have write permission, it raises an exception

Ex over Arithmetic Error using try_except_else

```
try:
    a=10/0
    print( a )
except ArithmeticError:
        print("This statement is raising an exception")
else:
    print("Try once more")
```

Output:This statement is raising an exception

except: Block without Exception

- You can even specify an except block without mentioning any exception (i.e., except:). This type of except block if present should be the last one that can serve as a wildcard (when multiple except blocks are present). But use it with extreme caution, since it may mask a real programming error.
- In large software programs, may a times, it is difficult to anticipate all types of possible exceptional conditions. Therefore, the programmer may not be able to write a different handler (except block) for every individual type of exception. In such situations, a better idea is to write a handler that would catch all types of exceptions. The syntax to define a handler that would catch every possible exception from the try block is,

```
try:
   Write the operations here
   ....................
except:
   If there is any exception, then execute this block.
   ....................
else:
   If there is no exception then execute this block.
```

Except with no Exception:

Except statement can also be used without specifying Exception.

```
try:
    a=10/0
    print(a)
except:
    print("This statement is raising an exception")
else:
    print("Iam Done executing")
```

Output: This statement is raising an exception

```
try:
      a=10/0
except IOError:
    print("Error occured during Input...Program Termianting")
except ValueError:
    print("Could not convert data to an Integer")
except:
    print("Unexpected Error...Prgram Terminating")
```

```
>>>
Unexpected Error...Prgram Terminating
>>>
```

try….except

- Catches all exceptions that occur
- It is not considered as good programming practice though it catches all exceptions as it does help the programmer in identifying the root cause of the problem that may occur
- **Syntax:**

```
try:
        you do your operations here;
    ……….
except Exception A:
        If there is ExceptionA, then execute this block
except Exception B:
        If there is ExceptionB, then execute this block
……….
```

Ex:

```
try:
    fh=open("testfile.txt","r")
    fh.write("This is my test file for exception")
    fh.close()
except IOError:
    print("Error: can't find file or read data")
```

Note: Try to write to the file when you do not have write permission, it raises an exception

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

Try....finally

- The try statement in Python can have an optional finally clause. This clause is executed and is generally used to release external resources.
- For example, we may be connected to a remote data centre through the network or working with a file or working with a Graphical User Interface (GUI).
- In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution.

Example:

```
try:
   f = open("test.txt",”r”")
   # perform file operations
finally:
   f.close()
```

This type of construct makes sure the file is closed even if an exception occurs.

try…..finally

- finally block is a place to put any code that must execute irrespective of try-block raised an exception or not
- else block can be used with finally block

- **Syntax:**

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

Ex:

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

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

- **Ex:**

```
try:
    a=10/0;
    print ("Exception occurred")
finally:
    print ("Code to be executed")
```

- Output: Code to be executed

Traceback (most recent call last):

File "C:/Users/RAVI@RGUKT/Desktop/try_except_else.py", line 2, in <module>

a=10/0

ZeroDivisionError: division by zero

Multiple Except Blocks

- Python allows you to have multiple except blocks for a single try block.
- The block which matches with the exception generated will get executed.
- A try block can be associated with more than one except block to specify handlers for different exceptions.
- However, only one handler will be executed. Exception handlers only handle exceptions that occur in the corresponding try block.
- We can write our programs that handle selected exceptions.
- The syntax for specifying multiple except blocks for a single try block can be given as,

- The syntax for specifying multiple except blocks for a single try block can be given as,

```
try:
   operations are done in this block
   ......................
except Exception1:
   If there is Exception1, then execute this block.
except Exception2:
   If there is Exception2, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Declaring Multiple Exception

- Multiple Exceptions can be declared using the same except statement:
- **Syntax:**

try:

code

except Exception1,Exception2,Exception3,..,ExceptionN

execute this code **in** case any Exception of these occur.

else:

execute code **in** case no exception occurred.

- Ex:

```
try:
    a=10/0
except IOError:
    print("IO Error found")
except ArithmeticError:
    print ("Arithmetic Exception")
else:
    print ("Successfully Done")
```

Output: Arithmetic Exception

```
try:
    num=int(input("Enter the number:"))
    print(num**2)
except (KeyboardInterrupt):
    print( "You should enter a number...Program Terminating")
except (ValueError):
    print("Please check before entering...Pragram Termianting")
```

```
Enter the number:10
100

>>> ========== RESTART =======
>>>
Enter the number:x
Please check before entering...Pragram Termianting
...
```

```
try:
    num=int(input("Enter the number:"))
    print(num**2)
except(KeyboardInterrupt,ValueError,TypeError):
    print("Please check...Program Terminating")
print("Bye")
```

```
>>>
Enter the number:q
Please check...Program Terminating
Bye
>>>
```

Raising Exceptions

- You can deliberately raise an exception using the **raise keyword**. The general syntax for the raise statement is,

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

- Here, Exception is the name of exception to be raised (example, TypeError).
- *args* is optional and specifies a value for the exception argument. If args is not specified, then the exception argument is None.
- The final argument, *traceback*, is also optional and if present, is the traceback object used for the exception.

```
try:
    num=20
    print(num)
    raise ValueError
except:
    print("Exception Raised...Program Terminating")
20
Exception Raised...Program Terminating
>>>
```

- **Ex:**

```
try:
    a=10
    print (a)
    raise NameError("Hello")
except NameError as e:
        print ("An exception occurred")
        print (e)
```

- **Output:**

```
10
An exception occurred
Hello
```

Explanation:

i) To raise an exception, raise statement is used. It is followed by exception class name.

ii) Exception can be provided with a value that can be given in the parenthesis. (here, Hello)

iii) To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

Re-raising Exception

- Python allows programmers to re-raise an exception. For example, an exception thrown from the try block can be handled as well as re-raised in the except block using the keyword raise.

```
try:
    f=open("File12.txt")  # Opening a non-exising file
except:
    print("File doesnot exist")
    raise   # re-raise the caught exception
```

```
File doesnot exist
Traceback (most recent call last):
  File "C:/Users/user/Desktop/E_H_Errors.py", line 166, in <module>
    f=open("File12.txt")  # Opening a non-exising file
FileNotFoundError: [Errno 2] No such file or directory: 'File12.txt'
>>>
```

Example

```
# import module sys to get the type of exception

import sys
while True:
   try:
      x = int(input("Enter an integer: "))
      r = 1/x
      break
   except:
      print("Oops!",sys.exc_info()[0],"occured.")
      print("Please try again.")
      print()
print("The reciprocal of",x,"is",r)
```

Output:

```
Enter an integer: a
Oops! <class 'ValueError'> occured.
Please try again.
Enter an integer: 1.3
Oops! <class 'ValueError'> occured.
Please try again.
Enter an integer: 0
Oops! <class 'ZeroDivisionError'>
occured.
Please try again.
Enter an integer: 2
The reciprocal of 2 is 0.5
```

User Defined Exception/Custom Exception

- Users can define their own exception by creating a new class in Python. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived form this class.

Example:

```
>>> class CustomError(Exception):
...     pass
...
```

- User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise.
- Most implementations declare a custom base class and derive others exception classes from this base class

- **Ex:**

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)
try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

- **Output: Received error: 2000**

```
# define Python user-defined exceptions
class Error(Exception): #Base class for other exceptions
   pass
class ValueTooSmallError(Error): #Raised when the input value is too small
   pass
class ValueTooLargeError(Error):#Raised when the input value is too large
   pass
# our main program
# user guesses a number until he/she gets it right
# you need to guess this number
number = 10
while True:
   try:
      i_num = int(input("Enter a number: "))
      if i_num < number:
         raise ValueTooSmallError
      elif i_num > number:
         raise ValueTooLargeError
      break
   except ValueTooSmallError:
      print("This value is too small, try again!")
      print()
   except ValueTooLargeError:
      print("This value is too large, try again!")
      print()
print("Congratulations! You guessed it correctly.")
```

Output:

```
Enter a number: 12
This value is too large, try again!
Enter a number: 0
This value is too small, try again!
Enter a number: 8
This value is too small, try again!
Enter a number: 10
Congratulations! You guessed it correctly.
```

Example

```
class ShortInputException(Exception):
    '''A user-defined exception class.'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast

try:
    text = input('Enter something --> ')
    if len(text) < 3:
        raise ShortInputException(len(text), 3)
    # Other work can continue as usual here
except EOFError:
    print('Why did you do an EOF on me?')
except ShortInputException as ex:
    print(('ShortInputException: The input was ' +
           '{0} long, expected at least {1}')
          .format(ex.length, ex.atleast))
else:
    print('No exception was raised.')
```

Output

```
$ python exceptions_raise.py
Enter something --> a
ShortInputException: The input was 1 long, expectedat
   least 3

$ python exceptions_raise.py
Enter something --> abc
No exception was raised.
```

How it works

- Here, we are creating our own exception type. This new exception type is called ShortInputException. It has two fields - length which is the length of the given input, and atleast which is the minimum length that the program was expecting.

- In the except clause, we mention the class of error which will be stored as the variable name to hold the corresponding error/exception object.

- This is analogous to parameters and arguments in a function call. Within this particular except clause, we use the length and atleast fields of the exception object to print an appropriate message to the user.

For Details Contact Me @ :

9247448766

ravikanth27787@gmail.com

python

Programming Language