UNIT – V (Chapter-2)

Error and Exception Handling

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.

But many a times, we come across some peculiar problems which are often categorized as exceptions. 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 also be categorized as synchronous and asynchronous exceptions. While 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.

Syntax and Logic 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.**

```
>>> i=0
>>> if i == 0 print(i)
SyntaxError: invalid syntax
```

Logic error **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*****.**

Exceptions

Even if a statement is syntactically correct, it may still cause an error when executed. Such errors that occur at run-time (or during execution) are known as ***exceptions***. An exception is an event, which occurs during the execution of a program and disrupts the normal flow of the program's instructions. When a program encounters a situation which it cannot deal with, it raises an exception. Therefore, we can say that an exception is a Python object that represents an error.

When a program raises an exception, it must handle the exception or the program will be immediately terminated. You can handle exceptions in your programs to end it gracefully, otherwise, if exceptions are not handled by programs, then error messages are generated..

Handling Exceptions

We can handle exceptions in our program by using try block and except block. A critical operation which can raise exception is placed inside the try block and the code that handles exception is written in except block. The syntax for try–except block can be given as,

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

Example:

```
print("x = ",x)     #NameError
try:
    print("x = ",x)

except:
    print("x variable is not defined..")
```

Handling Exceptions

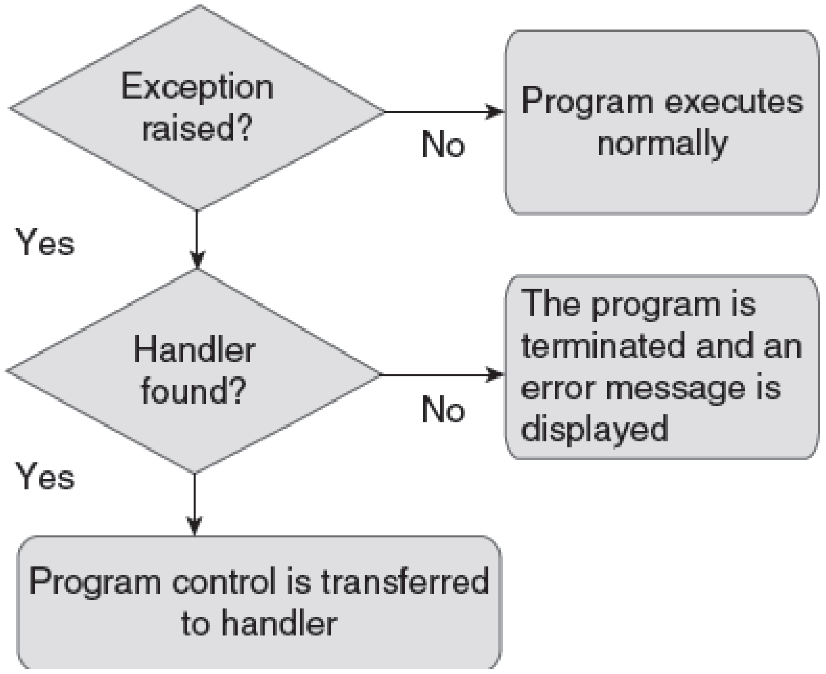
Example:

```
print("x = ",x)      #NameError
try:
    print("x = ",x)

except:
    print("x variable is not defined..")
```

Handling Exceptions

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


Exception raised?
No
Program executes normally
Yes
Handler found?
No
The program is terminated and an error message is displayed
Yes
Program control is transferred to handler

Example:

```
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")
```

OUTPUT

```
Enter the numerator : 10
Enter the denominator : 0
Denominator cannot be zero
```

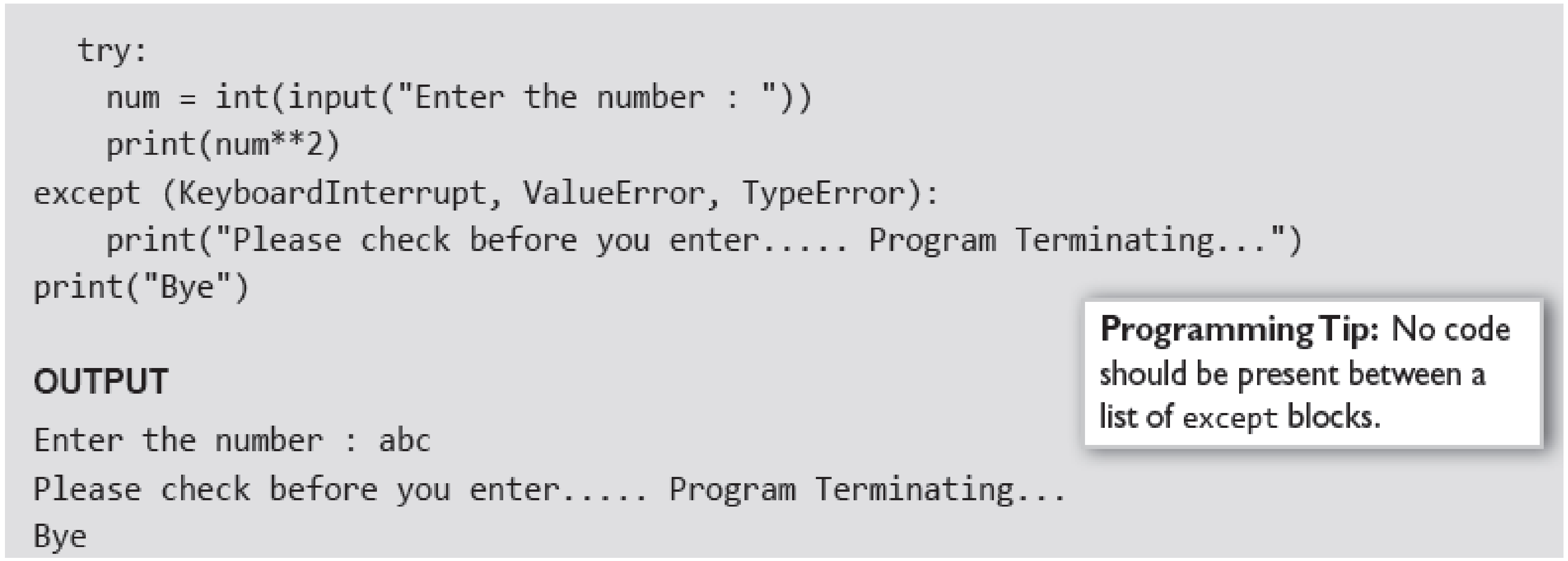
Multiple Except Blocks

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,

```
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.
```

```
try:
    print(x=10)
except NameError:
    print("x is not defined")
except:
    print("something else is wrong")
```

Multiple Exceptions in a Single Block — Example



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

OUTPUT

```
Enter the number : abc
Please check before you enter..... Program Terminating...
Bye
```

Programming Tip: No code should be present between a list of except blocks.

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 Block Without Exception — Example

```
try:
    file = pen('File1.txt')
    str = f.readline()
    print(str)
except IOError:
    print("Error occured during Input ...... Program Terminating...")
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error.... Program Terminating...")
```

Programming Tip: When an exception occurs, it may have an associated value, also known as the exception's *argument*.

OUTPUT

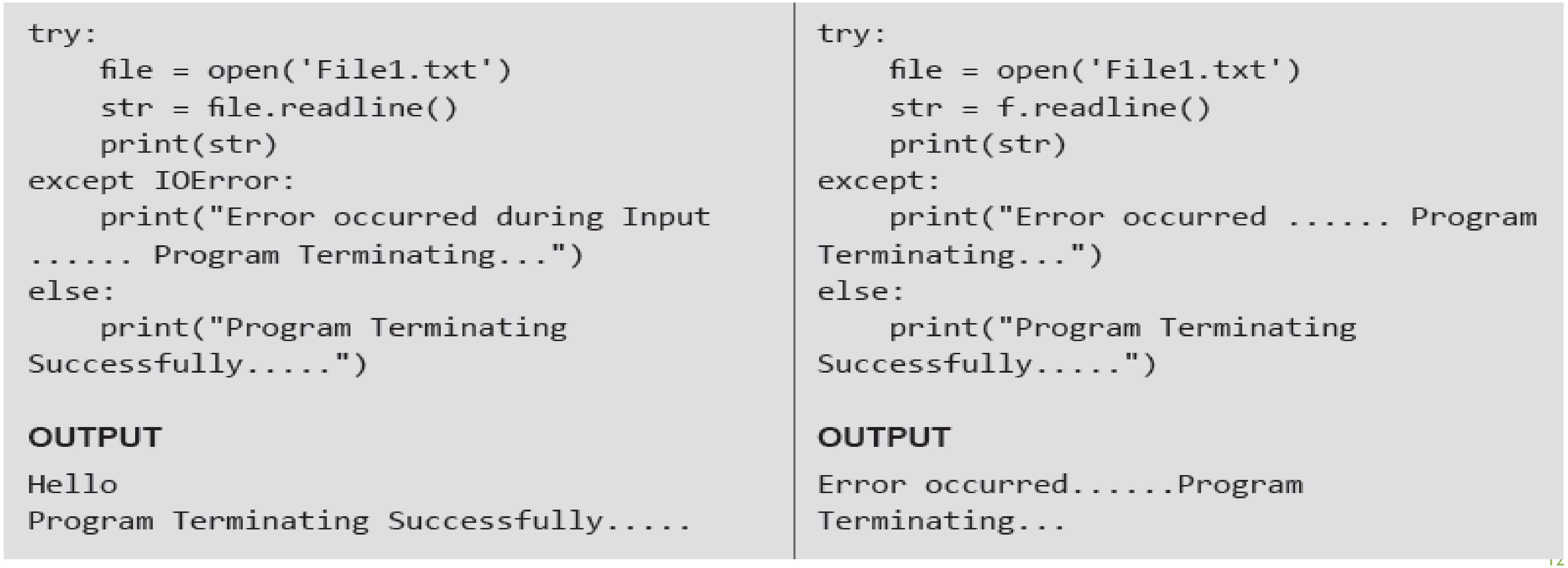
```
Unexpected error.... Program Terminating...
```

The Else Clause

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.

Examples:



```
try:
    file = open('File1.txt')
    str = file.readline()
    print(str)
except IOError:
    print("Error occurred during Input
...... Program Terminating...")
else:
    print("Program Terminating
Successfully.....")
```

OUTPUT

```
Hello
Program Terminating Successfully.....
```

```
try:
    file = open('File1.txt')
    str = f.readline()
    print(str)
except:
    print("Error occurred ...... Program
Terminating...")
else:
    print("Program Terminating
Successfully.....")
```

OUTPUT

```
Error occurred......Program
Terminating...
```

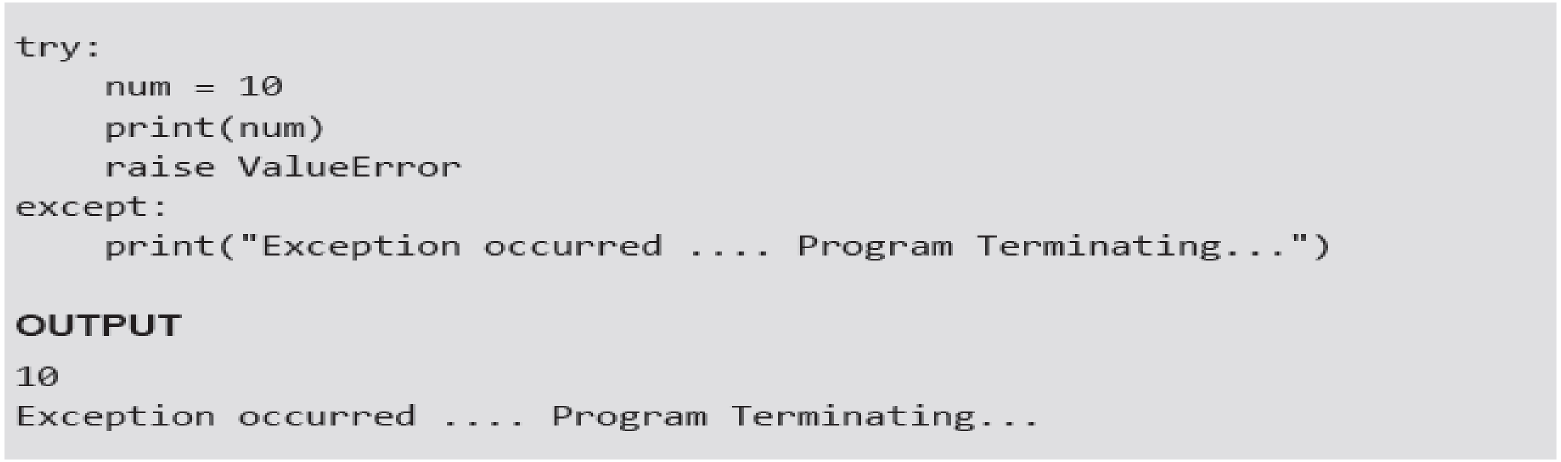
Raising Exceptions

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.

Example:

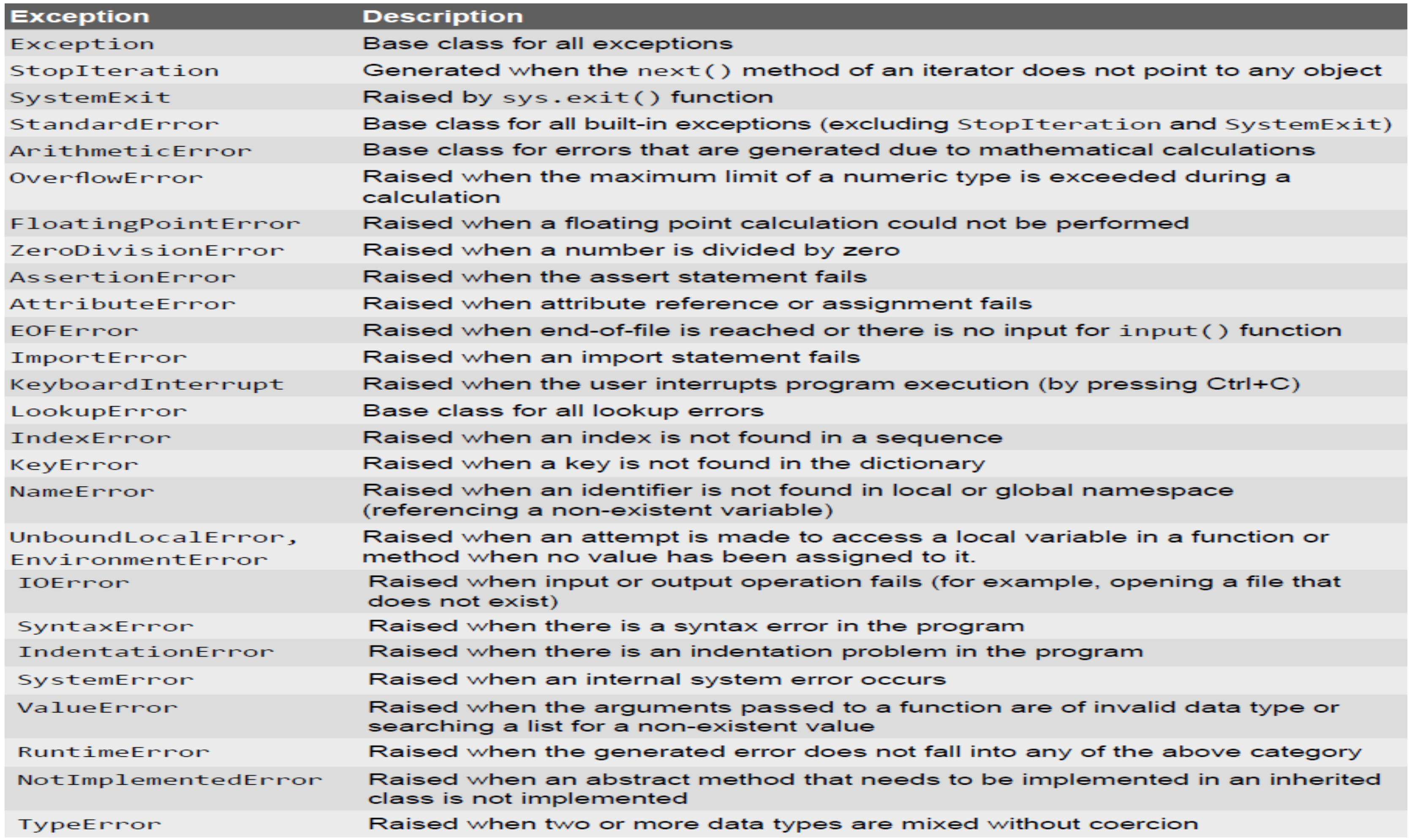


```
try:
    num = 10
    print(num)
    raise ValueError
except:
    print("Exception occurred .... Program Terminating...")
```

OUTPUT

```
10
Exception occurred .... Program Terminating...
```

Built-in and User-defined Exceptions



Exception	Description
Exception	Base class for all exceptions
StopIteration	Generated when the next() method of an iterator does not point to any object
SystemExit	Raised by sys.exit() function
StandardError	Base class for all built-in exceptions (excluding StopIteration and SystemExit)
ArithmeticError	Base class for errors that are generated due to mathematical calculations
OverflowError	Raised when the maximum limit of a numeric type is exceeded during a calculation
FloatingPointError	Raised when a floating point calculation could not be performed
ZeroDivisionError	Raised when a number is divided by zero
AssertionError	Raised when the assert statement fails
AttributeError	Raised when attribute reference or assignment fails
EOFError	Raised when end-of-file is reached or there is no input for input() function
ImportError	Raised when an import statement fails
KeyboardInterrupt	Raised when the user interrupts program execution (by pressing Ctrl+C)
LookupError	Base class for all lookup errors
IndexError	Raised when an index is not found in a sequence
KeyError	Raised when a key is not found in the dictionary
NameError	Raised when an identifier is not found in local or global namespace (referencing a non-existent variable)
UnboundLocalError, EnvironmentError	Raised when an attempt is made to access a local variable in a function or method when no value has been assigned to it.
IOError	Raised when input or output operation fails (for example, opening a file that does not exist)
SyntaxError	Raised when there is a syntax error in the program
IndentationError	Raised when there is an indentation problem in the program
SystemError	Raised when an internal system error occurs
ValueError	Raised when the arguments passed to a function are of invalid data type or searching a list for a non-existent value
RuntimeError	Raised when the generated error does not fall into any of the above category
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not implemented
TypeError	Raised when two or more data types are mixed without coercion

The finally Block

The try block has another optional block called finally which is used to define clean-up actions that must be executed under all circumstances. The finally block is always executed before leaving the try block. This means that the statements written in finally block are executed irrespective of whether an exception has occurred or not. The syntax of finally block can be given as,

```
try:
        Write your operations here
        ......................
        Due to any exception, operations written here will be skipped
finally:
        This would always be executed.
        ......................
```

Example:

```
try:
    print("Raising Exception.....")
    raise ValueError
finally:
    print("Performing clean up in Finally......")
```

OUTPUT

```
Raising Exception.....
Performing clean up in Finally......
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 4, in <module>
    raise ValueError
ValueError
```