Python Exception Handling

Exception handling is the process of responding to unwanted or unexpected events when a computer program runs. Exception handling allows you to separate error-handling code from normal code.

Error in Python can be of two types

Errors are the problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Two types of Error in python.

- **Syntax errors**

  Syntax errors, also known as parsing errors, When the proper syntax of the language is not followed then a syntax error is thrown.

- **Logical errors (Exceptions)**

  When in the runtime an error that occurs after passing the syntax test is called exception or logical type.

  Example: divide any number by zero then the ZeroDivisionError exception is raised, or when we import a module that does not exist then ImportError is raised.

  common built-in exceptions are other than above mention exceptions are:

Exception	Description
IndexError	When the wrong index of a list is retrieved.
AssertionError	It occurs when the assert statement fails
AttributeError	It occurs when an attribute assignment is failed.
ImportError	It occurs when an imported module is not found.
KeyError	It occurs when the key of the dictionary is not found.
NameError	It occurs when the variable is not defined.
MemoryError	It occurs when a program runs out of memory.

Exception	Description
TypeError	It occurs when a function and operation are applied in an incorrect type.

Error Handling

When an error and an exception are raised then we handle that with the help of the Handling method.

Handling Exceptions with Try/Except/Finally

We can handle errors by the Try/Except/Finally method. we write unsafe code in the try, fall back code in except and final code in finally block.

try:

You do your operations here

.....

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.

- 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.
- 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.

For 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()
```

with TRY/Finaly Block

```
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")
```

User Defined Exceptions

Exceptions need to be derived from the Exception class, either directly or indirectly. Although not mandatory, most of the exceptions are named as names that end in “Error” similar to the naming of the standard exceptions in python. For example,

```
class MyError(Exception):
    # Constructor or Initializer
   def __init__(self, value):
      self.value = value
    # __str__ is to print() the value
   def __str__(self):
      return(repr(self.value))
 try:
   raise(MyError(3*2))
 # Value of Exception is stored in error
except MyError as error:
   print('A New Exception occurred: ', error.value)
```

```
a=10
b=0
try:
  print ("This is outer try block")
  try:
    print (a/b)
```

```
    except ZeroDivisionError:
        print ("Division by 0")
    finally:
        print ("inside inner finally block")

except Exception:
    print ("General Exception")
finally:
    print ("inside outer finally block")
```