Python Exceptions

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.

What is an Exception?

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Exceptions versus Syntax Errors

When the interpreter identifies a statement that has an error, syntax errors occur. Consider the following scenario:

Code

```
#Python code after removing the syntax error

string = "Python Exceptions"


for s in string:

   if (s != o:

      print( s )
```

Code

```
#Python code after removing the syntax error

string = "Python Exceptions"


for s in string:

   if (s != o):

      print( s )
```

Different types of exceptions in python:

In Python, there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

SyntaxError: This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.

TypeError: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.

NameError: This exception is raised when a variable or function name is not found in the current scope.

IndexError: This exception is raised when an index is out of range for a list, tuple, or other sequence types.

KeyError: This exception is raised when a key is not found in a dictionary.

ValueError: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.

AttributeError: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.

IOError: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.

ZeroDivisionError: This exception is raised when an attempt is made to divide a number by zero.

ImportError: This exception is raised when an import statement fails to find or load a module.

These are just a few examples of the many types of exceptions that can occur in Python. It's important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from crashing.

Python Try Except

- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The else block lets you execute code when there is no error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the try statement:

Example

The try block will generate an exception, because x is not defined:

```
try:
  print(x)
except:
  print("An exception occurred")
```

Many Exceptions

You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

Example

Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
```

```
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Raise an exception

As a Python developer you can choose to throw an exception if a condition occurs.

To throw (or raise) an exception, use the raise keyword.

Example

Raise an error and stop the program if x is lower than 0:

```
x = -1
if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

The raise keyword is used to raise an exception.

You can define what kind of error to raise, and the text to print to the user.

Example

Raise a TypeError if x is not an integer:

```
x = "hello"
if not type(x) is int:
  raise TypeError("Only integers are allowed")
```

Try and Except Statement - Catching Exceptions

Code

Python code to catch an exception and handle it using try and except code blocks

```
a = ["Python", "Exceptions", "try and except"]
try:
    for i in range( 4 ):
       print( "The index and element from the array is", i, a[i] )
#if an error occurs in the try block, then except block will be executed by the Python interpreter
except:
   print ("Index out of range")
```

How to Raise an Exception

Code

#Python code to show how to raise an exception in Python

```
num = [3, 4, 5, 7]
if len(num) > 3:
   raise Exception( f"Length of the given list must be less than or equal to 3 but is {len(num)}" )
```

Try with Else Clause

Code

```
# Python program to show how to use else clause with try and except clauses


# Defining a function which returns reciprocal of a number
def reciprocal( num1 ):
   try:
      reci = 1 / num1
   except ZeroDivisionError:
      print( "We cannot divide by zero" )
   else:
      print ( reci )
# Calling the function and passing values
reciprocal( 4 )
reciprocal( 0 )
```

Finally Keyword in Python

The finally keyword is available in Python, and it is always used after the try-except block. The finally code block is always executed after the try block has terminated normally or after the try block has terminated for some other reason.

Here is an example of finally keyword with try-except clauses:

Code

```
# Python code to show the use of finally clause
# Raising an exception in try block
try:
   div = 4 // 0
   print( div )
# this block will handle the exception raised
except ZeroDivisionError:
   print( "Atepting to divide by zero" )
# this will always be executed no matter exception is raised or not
finally:
   print( 'This is code of finally clause' )
```

User-Defined Exceptions

Code

```
class EmptyError( RuntimeError ):
  def __init__(self, argument):
    self.arguments = argument
```

Once the preceding class has been created, the following is how to raise an exception:

Code

```
var = " "
try:
   raise EmptyError( "The variable is empty" )
except (EmptyError, var):
   print( var.arguments )
```