Topic: Exception Handling

Notebook:	Python_1st Yr syllabus		
Created:	5/6/2020 6:07 AM	**Updated:**	5/6/2020 1:25 PM
Author:	priya.agrawal8421@gmail.com		

Evernote

Topic: Exception Handling
Course: B.TECH CSE
Date:
Faculty: Priya Agrawal (T&D Department)

Link of Python Documentation

- FrontPage https://wiki.python.org/moin/

Objective :

- We want to catch all of the errors that could get generate. And as a programmer, We need to be as specific as possible. Therefore, Python allows programmers to deal with error efficiently.

List of errors (handle by the exception):

Exception	Description
AssertionError	Raised when the assert statement fails.
AttributeError	Raised on the attribute assignment or reference fails.
EOFError	Raised when the input() function hits the end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raised when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in the local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when a system operation causes a system-related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by the next() function to indicate that there is no further item to be returned by the iterator.

SyntaxError	Raised by the parser when a syntax error is encountered.
IndentationError	Raised when there is an incorrect indentation.
TabError	Raised when the indentation consists of inconsistent tabs and spaces.
SystemError	Raised when the interpreter detects internal error.
SystemExit	Raised by the sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of an 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.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translation.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of a division or module operation is zero.

What do you mean by the Exception ?

- Exception is the base class for all the exceptions in python.
- Events that are used to change/modify the flow of control through a program when the error occurs.
- Exceptions get triggered automatically on finding errors in Python.

Exception Hierarchy

6.1. Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

PTR's (Points to remember)

- Exceptions allow programmers to jump an exception handler in a single step, abandoning all function calls.

- You can relate like: **exceptions = optimized quality go-to statement**

i.e., the program error that occurs at runtime gets easily managed by the exception block. When the interpreter encounters an error, it lets the execution go to the exception part to solve and continue the execution instead of stopping.

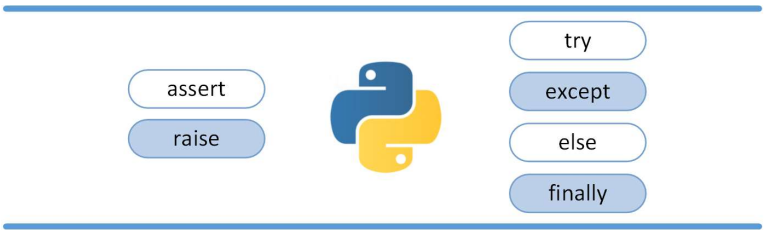
- While dealing with exceptions, the exception handler creates a mark & executes some code. Somewhere further within that program, the exception is raised that solves the problem & makes Python jump back to the marked location; by not throwing away/skipping any active functions that were called after the marker was left.

5 statements to processed the exceptions:

These exceptions are processed using five statements. These are:

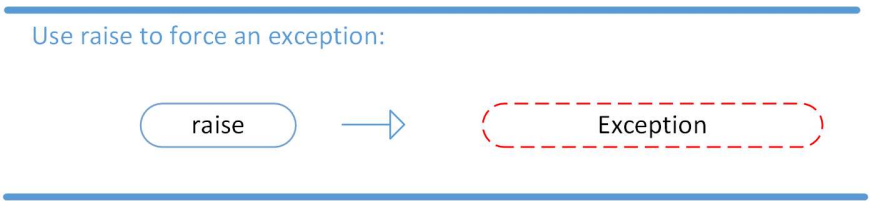
- ☑ **try/except**
- ☑ **try/finally**
- ☑ **assert**
- ☑ **raise**
- ☑ **with/as**

>>The last was an optional extension to Python 2.6 & Python 3.0.



Raising an Exception:

- Use raise to throw an exception if a condition occurs.
- If you want to throw an error when a certain condition occurs using raise, you could go about it like this:



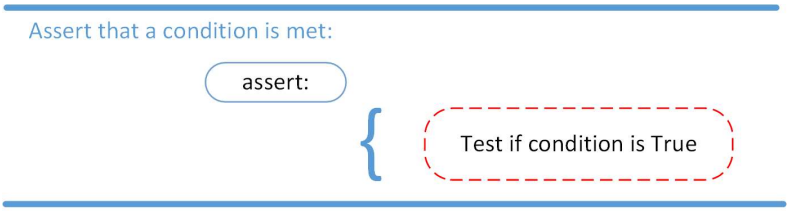

```
num=20
if num > 5:
   raise Exception('num should not exceed 5. The value of num was: {}'.format(num))
```

When you run this code, the output will be the following:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: num should not exceed 5. The value of num was: 20
```

The AssertionError Exception:

- Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We assert that a certain condition is met. If this condition turns out to be True, then that is excellent! The program can continue. If the condition turns out to be False, you can have the program throw an AssertionError exception.




```
import sys
assert ('linux' in sys.platform), "This code runs on Linux only."
```

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

When you run this code, the output will be the following:

```
Traceback (most recent call last):
  File "<input>", line 2, in <module>
AssertionError: This code runs on Linux only.
```

The program will come to halt and will not continue.

The try and except Block:

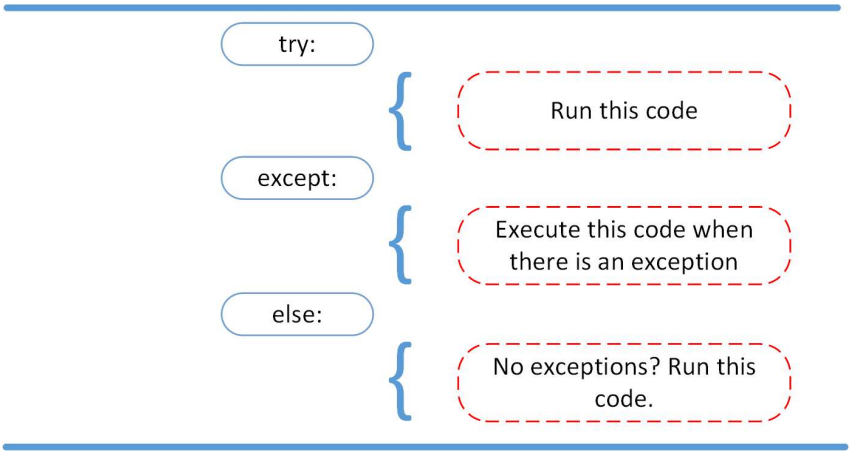
- The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a "normal" part of the program. The code that follows the except statement is the program's response to any exceptions in the preceding try clause.

```
num = 5
a=0
try:
   res = num/a
except ZeroDivisionError as e:
   r = e
   print (r)
```

Output: integer division or modulo by zero

The else Clause:

- In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.

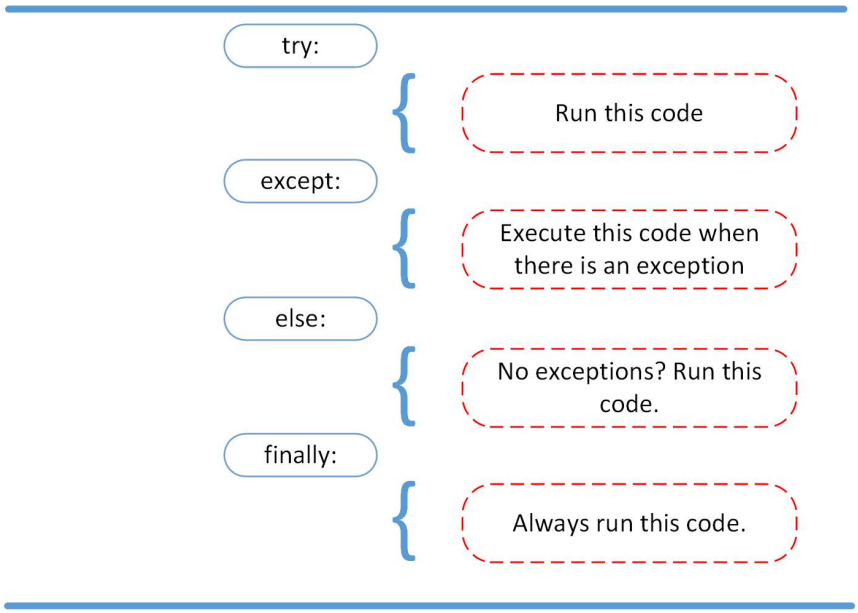



```
try:
   a = int(input("Enter a:"))
   b = int(input("Enter b:"))
   c = a/b;
   print("a/b = %d"%c)
except Exception:
   print("can't divide by zero")
else:
   print("Hi I am else block")
```

Output:
Enter a:10
Enter b:2
a/b = 5
Hi I am else block

Using finally (Cleaning Up):

- you always had to implement some sort of action to clean up after executing your code. Python enables you to do so clause.




```
try:
   # block of code
   # this may throw an exception
finally:
   # block of code
   # this will always be executed
```

```
try:
   fileptr = open("file.txt","r")
   try:
      fileptr.write("Hi I am good")
   finally:
      fileptr.close()
      print("file closed")
except:
   print("Error")
```

Output:
file closed
Error

raise allows you to throw an exception at any time.
assert enables you to verify if a certain condition is met and throw an exception if it isn't.
In the ***try clause***, all statements are executed until an exception is encountered.
except is used to catch and handle the exception(s) that are encountered in the try clause.
else lets you code sections that should run only when no exceptions are encountered in the try clause.
finally enables you to execute sections of code that should always run, with or without any previously encountered except

Roles of an Exception Handler in Python

- **Error handling:** The exceptions get raised whenever Python detects an error in a program at runtime. As a programmer, if you don't want the default behavior, then code a 'try' statement to catch and recover the program from an exception. Python will jump to the 'try' handler when the program detects an error; the execution will be resumed.
- **Event Notification:** Exceptions are also used to signal suitable conditions & then passing result flags around a program and text them explicitly.
- **Terminate Execution:** There may arise some problems or errors in programs that it needs a termination. So try/finally is used that guarantees that closing-time operation will be performed. The 'with' statement offers an alternative for objects that support it.
- **Exotic flow of Control:** Programmers can also use exceptions as a basis for implementing unusual control flow. Since there is no 'go to' statement in Python so that exceptions can help in this respect.

	EXAMPLES
Example_1	The 'try - Except' Clause with No Exception **try:** # all operations are done within this block. . . . . . . **except:** # this block will get executed if any exception encounters. . . . . . . **else:** # this block will get executed if no exception is found. . . . . . .
Example_2	The 'try - Finally' Clause **try:** # all operations are done within this block. . . . . . . # if any exception encounters, this block may get skipped. **finally:** . . . . . . # this block will definitely be executed.

Example_3	num = 5
	a=0
	try:
	  res = num/a
	except ZeroDivisionError:
	  print ("DIVIDED BY ZERO error occurs")
	
	Output: DIVIDED BY ZERO error occurs

Example_5	'except' Clause with Multiple Exceptions **try:** # all operations are done within this block. . . . . . . **except (Exception1 [, Exception2[,....Exception N]]]) :** # this block will get executed if any exception encounters from the above lists of exceptions. . . . . . . **else:** # this block will get executed if no exception is found. . . . . . .
Example_6	# Python program to handle simple runtime error

```
a = [1, 2, 3]
try:
    print "Second element = %d" %(a[1])

    # Throws error since there are only 3 elements in array
    print "Fourth element = %d" %(a[3])

except IndexError:
    print(''An error occurred'')
```

Output:
Second element = 2
An error occurred

Example_7

```
# Program to handle multiple errors with one except statement
try :
    a = 3
    if a < 4 :

        # throws ZeroDivisionError for a = 3

        b = a/(a-3)

    # throws NameError if a >= 4

    print ("Value of b = ", b)

except(ZeroDivisionError, NameError):
    print(\nError Occurred and Handled)
```

Output:
Error Occurred and Handled

Example_8

```
# Program to depict else clause with try-except
# Function which returns a/b
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print(a/b result in 0)
    else:
        print c

# Driver program to test above function
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output:
-5.0
a/b result in 0

Example_9

```
# Program to depict Raising Exception
```

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).
	
	try:
	  raise NameError("Hi there") # Raise Error
	except NameError:
	  print "An exception"
	  raise # To determine whether the exception was raised or not
	
	output:
	Traceback (most recent call last):
	  File "003dff3d748c75816b7f849be98b91b8.py", line 4, in
	  raise NameError("Hi there") # Raise Error
	NameError: Hi there

Example_10	f = open('missing')
	
	Output :
	Traceback (most recent call last):
	File "", line 1, in
	FileNotFoundError: [Errno 2] No such file or directory: 'miss
	
	Handle this error like this...
	
	try:
	  f = open('missing')
	  except OSError:
	  print('It failed')
	  except FileNotFoundError:
	  print('File not found')
	
	Output :
	Failed