Exception Handling in Python

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses **try** and **except** keywords to handle exceptions. Both keywords are followed by indented blocks.

```
try :
    #statements in try block
except :
    #executed when error in try block
```

The try: block contains one or more statements which are likely to encounter an exception. If the statements in this block are executed without an exception, the subsequent except: block is skipped.

If the exception does occur, the program flow is transferred to the except: block. The statements in the `except:` block are meant to handle the cause of the exception appropriately. For example, returning an appropriate error message.

You can mention a specific type of exception in front of the except keyword. The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

The rest of the statements after the except block will continue to be executed, regardless if the exception is encountered or not.

The following example will throw an exception when we try to devide an integer by a string.

Example: try...except blocks

```
try:
    a=5
    b='0'
    print(a/b)
except:
    print('Some error occurred.')
print("Out of try except blocks.")
```

Result:

```
Some error occurred.

Out of try except blocks.
```

You can mention a specific type of exception in front of the except keyword.

The subsequent block will be executed only if the specified exception occurs. There may be multiple except clauses with different exception types in a single try block. If the type of exception doesn't match any of the except blocks, it will remain unhandled and the program will terminate.

Example:

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

Result:

```
Unsupported operation

Out of try except blocks
```

As mentioned above, a single try block may have multiple except blocks. The following example uses two except blocks to process two different exception types:

Example:

```
try:
    a=5
    b=0
    print (a/b)
except TypeError:
    print('Unsupported operation')
except ZeroDivisionError:
    print ('Division by zero not allowed')
print ('Out of try except blocks')
```

Result:

```
Division by zero not allowed

Out of try except blocks
```

However, if variable b is set to '0', TypeError will be encountered and processed by corresponding except block.

else and finally

In Python, keywords **else** and **finally** can also be used along with the try and except clauses. While the except block is executed if the exception occurs inside the try block, the else block gets processed if the try block is found to be exception free.

```
try:
    #statements in try block
except:
```

```
    #executed when error in try block
else:
    #executed if try block is error-free
finally:
    #executed irrespective of exception occured or not
```

The finally block consists of statements which should be processed regardless of an exception occurring in the try block or not. As a consequence, the error-free try block skips the except clause and enters the finally block before going on to execute the rest of the code. If, however, there's an exception in the try block, the appropriate except block will be processed, and the statements in the finally block will be processed before proceeding to the rest of the code.

The example below accepts two numbers from the user and performs their division. It demonstrates the uses of else and finally blocks.

Example:

```
try:
    print("try block")
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

The first run is a normal case. The out of the else and finally blocks is displayed because the try block is error-free.

Result:

```
try block

Enter a number: 10

Enter another number: 2

else block

Division = 5.0

finally block

Out of try, except, else and finally blocks.
```

The econd run is a case of division by zero, hence, the except block and the finally block are executed, but the else block is not executed.

Result:

```
try block

Enter a number: 10

Enter another number: 0

except ZeroDivisionError block

Division by 0 not accepted

finally block

Out of try, except, else and finally blocks.
```

In the third run case, an uncaught exception occurs. The finally block is still executed but the program terminates and does not execute the program after the finally block.

Result:

```
try block

Enter a number: 10

Enter another number: xyz

finally block

Traceback (most recent call last):

  File "C:\python36\codes\test.py", line 3, in <module>

    y=int(input('Enter another number: '))

ValueError: invalid literal for int() with base 10: 'xyz'
```

Typically the finally clause is the ideal place for cleaning up the operations in a process. For example closing a file irrespective of the errors in read/write operations. This will be dealt with in the next chapter.

Raise an Exception

Python also provides the **raise** keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

Example: Raise an Exception

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
```

```
else:
    print(x, "is within the allowed range")
```

Result:

```
Enter a number upto 100: 200

200 is out of allowed range

Enter a number upto 100: 50

50 is within the allowed range
```

```
while True:
    try:
        n = input("Please enter an integer: ")
        n = int(n)
        break
    except ValueError:
        print("No valid integer! Please try again ...")
print("Great, you successfully entered an integer!")
```

Index Error

When you are trying to access an index (sequence) of a list that does not exist in that list or is out of range of that list, an index error is raised.

```
try:
    a = ['a', 'b', 'c']
    print (a[4])
except LookupError:
    print ("Index Error Exception Raised, list index out of range")
else:
    print ("Success, no error!")
Index Error Exception Raised, list index out of range
```

Name Error

Name Error is raised when a local or global name is not found.

In the below example, `ans` variable is not defined. Hence, you will get a `name error`.

```
try:
    print (ans)
except NameError:
    print ("NameError: name 'ans' is not defined")
else:
    print ("Success, no error!")
NameError: name 'ans' is not defined
```

Type Error

Type Error Exception is raised when two different or unrelated types of operands or objects are combined.

In the below example, an integer and a string are added, which results in a `type` error.

```
try:
    a = 5
    b = "DataCamp"
    c = a + b
except TypeError:
    print ('TypeError Exception Raised')
else:
    print ('Success, no error!')
TypeError Exception Raised
```

Value Error

Value error is raised when the built-in operation or a function receives an

argument that has a correct `type` but invalid `value`.

In the below example, the built-in operation `float` receives an argument, which is a sequence of characters (value), which is invalid for a type float.

```
try:
    print (float('DataCamp'))
except ValueError:
    print ('ValueError: could not convert string to float: \'DataCamp\'')
else:
    print ('Success, no error!')
ValueError: could not convert string to float: 'DataCamp'
```

number = int(input("Enter a number between 1 - 10"))

print("you entered number", number)

This program works perfectly fun as long as the user enters a number, but what happens if the user puts in something else (like a string)?

Enter a number between 1 - 10

hello

You can see that the program throws us an error when we enter a string.

Traceback (most recent call last):

File "enter_number.py", line 1, in

number = int(input("Enter a number between 1 - 10 "))

ValueError: invalid literal for int() with base 10: 'hello'

ValueError is an exception type. Let's see how we can use exception handling to fix the previous program

Let us try to access the array element whose index is out of bound and handle the corresponding exception.

```
# 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
```

Else Clause:

In python, you can also use else clause on try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

```
# 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)
```

The output for above program will be :

```
-5.0
a/b result in 0
```