July 26, 202

Date
Page

WEEK 11

Exception Handling

Error in python can be of two types i.e, Syntax Errors and Exceptions. 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.

DIFFERENCE BETWEEN SYNTAX ERROR AND EXCEPTIONS

Syntax Error : As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

EXAMPLE CODE →

```
amount = 10000
if (amount > 2999)
    print ('You are eligible for gift')
```

OUTPUT →

```
File "C:\Users\hp\.spider-py3\filename.py", line 2
    if (amount > 2999)
SyntaxError: invalid syntax
```

Exceptions : Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of program.

EXAMPLE CODE →
```
marks = 100
a = marks / 0
print (a)
```

OUTPUT →
```
Traceback (most recent call last):
  File "filename.py", line 2, in <module>
    a = marks / 0
ZeroDivisionError : division by zero
```

In the above example, the code raised the ~~Exception~~ ZeroDivisionError as we are trying to divide a number by 0.

[#Note : Exception is the base class for all the exceptions in Python.]

TRY & EXCEPT

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

CODE :

```
# Python program to handle simple runtime error

a = [1, 2, 3]

try :
    print ("Second Element = ", a[1])

    #throws an error since there are only 3 elements in list
    print ("Fourth Element = ", a[3])

except IndexError:
    print ('An Error Occurred')
```

OUTPUT :

```
Second Element = 2
An Error Occurred
```

→ A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed.

CODE :

```
# Python program to handle multiple errors with one
# Handle statement

try :
    a = 3
    if a < 4 :
```

```
    b = a/(a-3)     # throws ZeroDivisionError for a = 3

    print('Value of b =', b)
# throws NameError for a >= 4

except (ZeroDivisionError, NameError):
    print ("Error Occurred & Handled")
```

OUTPUT :

Error Occurred & Handled

→ If you change the value of 'a' greater than equal to 4, the output will be :

Value of b =
Error Occurred & Handled

→ The output above is so because as soon as python tries to access the value of b, NameError occurs.

FINALLY KEYWORD

Python provides a keyword finally, which is always executed after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception.

Let us write a sample code :

```
# Python program to demonstrate finally

try :
    k = 5//0
    print (k)

except ZeroDivisionError :
    print ("can't divide by zero")

finally :
    # this block is always executed
    # regardless of exception generation
    print ('This is always executed')
```

OUTPUT :

```
can't divide by zero
This is always executed
```

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

Let's write a code to depict how this works :

CODE :

Program to depict how to raise exception.

```
a = int(input())
if a < 18 :
    raise Exception("You are underage, can't vote")
```

OUTPUT :

```
3
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    raise Exception("You are underage, can't vote")
Exception : You are underage, can't vote
```

One more Example Code :

```
try:
    raise NameError("Hi there")  # Raise Error

except NameError :
    print('An exception')
    raise # to determine whether exception was raised or not
```

OUTPUT :

```
An exception
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    raise NameError("Hi there") # Raise Error
NameError : Hi there
```

Date
July 27, 2021

classmate
Date
Page

Functional Programming

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on 'what to solve' in contrast to an imperative style where the main focus is 'how to solve'. It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.

PYTHON ITERATORS

→ An iterator is used to iterate through iterable objects.

→ In Python, iterable objects are objects that can be iterated upon, like lists, tuples, sets & even strings.

→ We have already learned how to iterate through an iterable object using the for loop.

→ Another way to iterate through an object is to use an iterator.

→ But before we can do that, first we need to create an iterator from an iterable object.

- The iter() function returns an iterator from an iterable object.

```
a = [1, 2, 3, 4]

iter_a = iter(a)

print(next(iter_a))
print(next(iter_a))
```

Output

```
1
2
```

→ next() function returns the next element in the iterator object.

→ It iterates and prints the elements one-by-one.

PYTHON GENERATORS

→ Python generators are simple way of creating iterators.

→ A generator is a function that returns an object (iterator) which we can iterate over (one value at a time)

→ To create a generator, we use yield statement instead of a return statement.

→ If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

- The difference is that while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

- Here is how a generator function differs from a normal function().

  - Generator function contains one or more yield statemen
  - When called, it returns an object (iterator) but does not start execution immediately.
  - Methods like iter() & next() are implemented automatically. So we can iterate through the items using next().
  - Once the function yields, the function is paused and the control is transferred to the caller.
  - Local variable and their states are remembered between successive ~~called~~ calls.
  - Finally, when the function terminates, StopIteration is raised automatically on further calls.

- Let us see the working of a generator with help of an example code:

```
def square (limit):
    x = 0
    while x < limit :
        yield x * x
        yield x * x * x
        x += 1

a = square (5)
print ( next (a), next (a))
print ( next (a), next (a))
print (next (a), next (a))
```

Output

```
0 0
1 1
4 8
```

INLINE STATEMENTS

(1) Inline If

- Inline if is a concise version of if...else statement that can be written in just one line.

WAYS TO USE PYTHON INLINE IF STATEMENT

- Inline if Without else
- with else statement
- Inline if...elif

(a) Inline if without else

SYNTAX : if <condition> : <statement>

Example : a = True
if a : print ('True')

(b) Inline if with else statement

SYNTAX : <statement 1> if <condition> else <statement 2>

Example : color = blue
a = 'Color is red' if color == red else 'Blue'
print (a)

Example : d = 4
print ('Greater' if d > 3 else 'smaller')

(c) Inline if ... elif

SYNTAX : <statement 1> if <condition> else <statement 2> if <condition>
else <statement 3>

Example :

value = 10
print ('Less than 10' if value < 10 else 'More than 10' if value > 10
else 'Equal to 10')

(2) Inline while loop

Usual Code:

```
a = 5
while a > 0 :
    print(a)
    a -= 1
```

Inline Code:

```
a = 5
while a > 0 : print(a); a -= 1
```

LIST COMPREHENSION (inline for loop)

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example: Based on a list of fruits, you want a new list, containing only the fruits with the letter 'a' in the name.

Without list comprehension you will have to write a for statement with a conditional test inside.

CODE:

```
fruits = ['apple', 'banana', 'cherry', 'kiwi', 'mango']
newlist = []
for x in fruits:
    if 'a' in x:
        newlist.append(x)

print(newlist)
```

→ With list comprehension you can do all that with only one line of code.

CODE:

```
fruits = ['apple', 'banana', 'kiwi', 'cherry', 'mango']

newlist = [x for x in fruits if 'a' in x]

print (newlist)
```

OUTPUT:

```
['apple', 'banana', 'mango']
```

The SYNTAX

```
newlist = [expression for item in iterable if condition == Tr
```

→ The return value is a new list, leaving the old list unchanged.

Condition :→ It is like a filter that only accepts the items that valuate to True.
→ The condition is optional & can be ~~ommitie~~ ommitted.

Iterable :→ It can be any iterable object like list, tuple etc.
→ You can use range() function to create an iterable.

```
newlist = [x for x in range(10)]
```

Expression : It is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

```
newlist = [x.upper() for x in fruits]
```

→ You can set the outcome whatever you like :

```
newlist = ['hello' for x in fruits]
```

→ The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome :

```
newlist = [x if x != 'banana' else 'orange' for x in fruits]
```

LAMBDA FUNCTION.

In Python, anonymous functions means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

SYNTAX: lambda <arguments> : <expression>

(1) The function can have any number of arguments but only one expression, which evaluated and returned.

(2) One is free to use lambda functions wherever function objects are required.

(3) You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.

(4) EXAMPLE :

```
cube = lambda x : x*x*x
print(cube(7))
>> 343

add = lambda x,y : x+y
print(add(4,3))
>> 7
```

ENUMERATE ( )

A lot of times when dealing with iterators, we also get a need to keep a count of iterations. Python eases the programmer's task by providing a built-in function enumerate ( ) for this task.

Enumerate( ) method adds a counter to an iterable and returns it in a form of enumerate object. This enumerate object can be then used directly in for loops or be converted into a list of tuples using list ( ) method.

SYNTAX : `enumerate (iterable , start=0)`

PARAMETERS :
Iterable - any object that supports iteration
Start - the index value from which the counter is to be started, by default it is 0.

CODE :

```
l1 = ['eat', 'sleep', 'repeat']

for x in enumerate (l1):
    print (x)

#changing index & printing separately
for count, x in enumerate (l1, 100):
    print (count, x)
```

OUTPUT :

```
(0, 'eat')
(1, 'sleep')
(2, 'repeat')
100 eat
101 sleep
102 repeat
```

ZIP()

The purpose of zip() method is to map the similar index of multiple containers so that they can be used just using as single entity.

CODE:

```
fruits = ['apple', 'mango', 'banana', 'cherry']
size = [5, 5, 6, 6]

print(list(zip(fruits, size)))

print(dict(zip(fruits, size)))
```

OUTPUT:

```
[('mango', [('apple', 5), ('mango', 5), ('banana', 6),
('cherry', 6)]

{'apple': 5, 'mango': 5, 'banana': 6, 'cherry': 6}
```

MAP()

Map() function returns a list of the results after applying the given function to each item of the given iterable (list, tipple)

SYNTAX: map(function, iterable)

PARAMETERS :

function → it is a fn to which map passes each element of given iterable

iterable → it is an iterable which is to be mapped

Return Type → returns an iterator of map class.

CODE :

```
def addition (n):
    return n+n

#we double all numbers using map()
numbers = (1, 2, 3, 4)
results = map (addition, numbers)
print (results)  # does not print values

#for printing value
for result in results:
    print (result, end=' ')
```

OUTPUT :

```
<map object at 0x7fae3004b630>
2 4 6 8
```

→ Let us look at one more code to understand the functionality of map().

CODE :

```
a = [10, 20, 30, 40, 50]
b = [1, 2, 3, 4, 5]

def sub(x, y):
    return x-y

c = map(sub, a, b)
print(list(c))
```

OUTPUT :

[9, 18, 27, 36, 45]

FILTER()

The filter() method filters the given sequence with the help of a function that tests each elements in the sequence to be true or not.

SYNTAX : filter (function , sequence)

PARAMETERS :

function → it tests if each element of a sequence is true or not.

sequence → sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.

Return type → returns an iterator that is already filtered

CODE:

```
# function that filter vowels
def func(variable):
    letters = ['a', 'e', 'i', 'o', 'u']

    if (variable in letters):
        return True
    else:
        return False

sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

filtered = filter(func, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

OUTPUT:

```
The filtered letters are:
e
e
```