**Loops Control Statements**

Loop control statements change the execution from its normal sequence.

When the execution leaves a scope, all automatic objects that were created in that scope are destroyed.

|  |  |  |
| --- | --- | --- |
| S.No. | Control Statement | Description |
| 1 | **break statement** | Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| 2 | **continue statement** | Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | **pass statement** | used when a statement is required syntactically but you do not want any command or code to execute. |

# break statement

* Used for premature termination of the current loop
* After abandoning the loop, execution at the next statement is resumed
* Break statement can be used in both while and for loops

#!/usr/bin/python3

for letter in 'Python':

if letter == 'h':

break

print ('Current Letter: ', letter)

Output:

Current Letter: P

Current Letter: y

Current Letter: t

# continue statement

* Returns the control to the beginning of the current loop
* When encountered, the loop starts next iteration without executing the remaining statements in the current iteration

#!/usr/bin/python3

for letter in 'Python':

if letter == 'h':

continue

print ('Current Letter: ', letter)

Output:

Current Letter: P

Current Letter: y

Current Letter: t

Current Letter: o

Current Letter: n

# pass statement

* It is used when a statement is required syntactically but you do not want any command or code to execute.
* The pass statement is a null operation; nothing happens when it executes.
* The pass statement is also useful in places where your code will eventually go, but has not been written yet i.e. in stubs.

#!/usr/bin/python3

for letter in 'Python':

if letter == 'h':

pass

print("This is pass block")

print ('Current Letter: ', letter)

Output:

Current Letter: P

Current Letter: y

Current Letter: t

This is pass block

Current Letter: h

Current Letter: o

Current Letter: n

# Iterator

Iterator is an object which allows to traverse through all the elements of a collection, regardless of its specific implementation.

In Python, an iterator object implements two methods, iter() and next().

Iterator object can be traversed using regular for statement

#!/usr/bin/python3

list1 = [1, 2, 3, 4, 5]

it = iter(list1)

print(it)

print(next(it))

print(it)

print(next(it))

for var in it:

print(var)

Output:

<list\_iterator object at 0x7ff6bc8c1588>

1

<list\_iterator object at 0x7ff6bc8c1588>

2

3

4

5

# Generator

A generator is a function that produces or yields a sequence of values using yield method.

When a generator function is called, it returns a generator object without even beginning execution of the function.

When the next() method is called for the first time, the function starts executing until it reaches the yield statement, which returns the yielded value.

The yield keeps track i.e. remembers the last execution and the second next() call continues from previous value.

Example

The following example defines a generator, which generates an iterator for all the Fibonacci numbers.

#!usr/bin/python3

import sys

def fibonacci(n): #generator function

a, b, counter = 0, 1, 0

while True:

if (counter > n):

return

yield a

a, b = b, a + b

counter += 1

f = fibonacci(5) #f is iterator object

while True:

try:

print (next(f), end=" ")

except StopIteration:

sys.exit()

Output:

0 1 1 2 3 5

# END