**Iteration**

Python scripts can implement looping, or iteration, with the **while** and **for** constructs. In each case a block of code is executed repeatedly until a loop exit condition is satisfied.

Repetition using **while** loop

A **while** loop repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

The syntax of a **while** loop in Python programming language is:

while expression:

statement(s)

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

A key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

count = 0

while (count < 9):

print 'The count is:', count

count = count + 1

print "Good bye!"

When the above code is executed, it produces the following result −

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

Repetition using **while** loop with else statement

Python supports an **else** statement with the **while** loop, the **else** statement is executed when the condition becomes false.

Example

count = 0

while count < 5:

print count, " is less than 5"

count = count + 1

else:

print count, " is not less than 5"

When the above code is executed, it produces the following result −

0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

5 is not less than 5

Repetition using **for** loop

The **for** loop iterates over the items of any sequence, such as a list or a string.

The syntax of a **for** loop in Python programming language is:

for iterating\_var in sequence:

statements(s)

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating\_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating\_var*, and the statement(s) block is executed until the entire sequence is exhausted.

Example

for letter in **'Python':** # First Example

print 'Current Letter :', letter

fruits = ['banana', 'apple', 'mango']

for fruit in **fruits**: # Second Example

print 'Current fruit :', fruit

print "Good bye!"

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : h

Current Letter : o

Current Letter : n

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

Repetition using **for** loop – by sequence index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example −

fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

print 'Current fruit :', fruits[index]

print "Good bye!"

When the above code is executed, it produces the following result −

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

Here, we took the assistance of the len() built-in function, which provides the total number of elements, as well as the range() built-in function to give us the actual sequence to iterate over.

range(stop)

* stop: Number of integers (whole numbers) to generate, starting from zero. eg. range(3) == [0, 1, 2].

Repetition using **for** loop with else statement

Python supports an **else** statement with the **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

Loop control statements

Loop control statements change execution from its normal sequence. Python supports the following control statements:

**break statement** - Terminates the loop statement and transfers execution to the statement immediately following the loop.

Example

for letter in 'Python':

if letter == 'h':

break

print 'Current Letter :', letter

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

[**continue statement**](http://www.tutorialspoint.com/python/python_continue_statement.htm)- rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

Example

for letter in 'Python':

if letter == 'h':

continue

print 'Current Letter :', letter

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

**pass statement** - 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** is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example):

for letter in 'Python':

if letter == 'h':

pass

print 'This is pass block'

print 'Current Letter :', letter

print "Good bye!"

When the above code is executed, it produces following result −

Current Letter : P

Current Letter : y

Current Letter : t

This is pass block

Current Letter : h

Current Letter : o

Current Letter : n

Good bye!

Nested loops

Python programming language allows you to use one loop inside another loop.

**nested for loop**:

for iterating\_var in sequence:

for iterating\_var in sequence:

statements(s)

statements(s)

**nested while loop**

while expression:

while expression:

statement(s)

statement(s)

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

i = 2

while(i < 100):

j = 2

while(j <= (i/j)):

if not(i%j): break

j = j + 1

if (j > i/j) : print i, " is prime"

i = i + 1

print "Good bye!"

When the above code is executed, it produces following result −

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

23 is prime

29 is prime

31 is prime

37 is prime

41 is prime

43 is prime

47 is prime

53 is prime

59 is prime

61 is prime

67 is prime

71 is prime

73 is prime

79 is prime

83 is prime

89 is prime

97 is prime

Good bye!

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.