**Loop**

A loop statement allows to execute a statement or group of statements multiple times. Python provides the following types of loops:

|  |  |  |
| --- | --- | --- |
| S.No. | Loop Type | Description |
| 1 | **while loop** | Repeats a statement or group of statements while a given condition is TRUE.  It tests the condition before executing the loop body. |
| 2 | **for loop** | Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | **nested loops** | One or more loop inside any another while, or for loop. |

# While loop

A while loop repeatedly executes target statement(s) as long as a given condition is true.

**Syntax**

while expression:

statement(s)

* The condition may be any expression, and true is any non-zero value.
* While loop tests the condition before executing the loop body.

#!/usr/bin/python3

count = 0

while (count < 5):

print ('Count is: ', count)

count = count + 1

print ("Loop Over!")

Output:

Count is: 0

Count is: 1

Count is: 2

Count is: 3

Count is: 4

Loop Over!

# For loop

for statement in Python has the ability to iterate over the items of any sequence, such as a list or a string.

**Syntax**

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.

#!/usr/bin/python3

for var in [1, 2, 3, 4, 5]:

print("Var: ", var)

print("Loop Over")

Output:

Var: 1

Var: 2

Var: 3

Var: 4

Var: 5

Loop Over

## Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself.

#!/usr/bin/python3

list1 = ['word1', 'word2', 'word3']

for index in range(len(list1)):

print("list[index]: ", list1[index])

print("End of list")

Output:

list[index]: word1

list[index]: word2

list[index]: word3

End of list

# Using else Statement with Loops

Python supports having an else statement associated with a loop statement.

* If the else statement is used with a while loop, the else statement is executed when the condition becomes false.
* If the else statement is used with a for loop, the else statement is executed when the loop has exhausted iterating the list.

#!/usr/bin/python3

count = 0

while count < 5:

print (count, " is less than 5")

count = count + 1

else:

print (count, " is not less than 5")

print("Loop Over")

Output:

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

Loop Over

#!/usr/bin/python3

for count in range(5):

print (count, " is less than 5")

count = count + 1

else:

print (count, " is not less than 5")

print("Loop Over")

Output:

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

Loop Over

# range() function

The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.

**class range(stop)**

**class range(start, stop[, step])**

* step defaults to 1
* If step is zero, ValueError is raised.
* start defaults to 0
* For a positive step, the contents of a range r are determined by the formula r[i] = start + step\*i where i >= 0 and r[i] < stop.
* For a negative step, the contents of the range are still determined by the formula r[i] = start + step\*i, but the constraints are i >= 0 and r[i] > stop.

Ranges do support negative indices, but these are interpreted as indexing from the end of the sequence determined by the positive indices.

Ranges containing absolute values larger than sys.maxsize are permitted but some features (such as len()) may raise OverflowError.

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> list(range(0, 30, 5))

[0, 5, 10, 15, 20, 25]

>>> list(range(0, 10, 3))

[0, 3, 6, 9]

>>> list(range(0, -10, -1))

[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

>>> list(range(0))

[]

>>> list(range(1, 0))

[]

## Advantage over a regular list or tuple

A range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).

Range objects implement the collections.abc.Sequence ABC, and provide features such as containment tests, element index lookup, slicing and support for negative indices.

>>> r = range(0, 20, 2)

>>> r

range(0, 20, 2)

>>> 11 in r

False

>>> 10 in r

True

>>> r.index(10)

5

>>> r[5]

10

>>> r[:5]

range(0, 10, 2)

>>> r[-1]

18

Two range objects are considered equal if they represent the same sequence of values. (Note that two range objects that compare equal might have different start, stop and step attributes, for example range(0) == range(2, 1, 3) or range(0, 3, 2) == range(0, 4, 2).)

**Changed in version 3.2:** Implement the Sequence ABC. Support slicing and negative indices. Test int objects for membership in constant time instead of iterating through all items.

**Changed in version 3.3:** Define ‘==’ and ‘!=’ to compare range objects based on the sequence of values they define (instead of comparing based on object identity).

# END