

எங்கள் வாழ்வும் எங்கள் வளமும்  
மங்காத தமிழ் என்று சங்கே முழங்கு ... புரட்சிக்கவி

## NOTICE

[www.DataScienceInTamil.com](http://www.DataScienceInTamil.com)

**Day 9 and 10 - Batch 3 - Python Language**

**Chapter 007 For loop, while, break, continue**

**Watch the video in below You Tube link**

<https://youtu.be/lQeJlUHeutA> - Day 9

<https://youtu.be/oqhGl2zE> Pg - Day 10

-----

**To watch the recorded Python and Data Science  
videos in YouTube:**

<https://www.youtube.com/channel/UCTCMjShTpZg96cXloCO9q1w>

**Official Website:**

<https://DataScienceInTamil.com/>

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

<https://www.DatascienceInTamil.com/#faq>

**To join DataScienceInTamil Telegram group:**

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

<https://t.me/joinchat/lUZEsr-zidpjZjEx>

**To Join the class, please fill the form :**

<https://forms.gle/QFpLHwAoinFaX2cE6>

**Join Zoom Meeting (From Sep 26 2022 to Oct 26  
2022)**

<https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFRUVpTzZ2S1Vsdz09>

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

-----

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.

- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at [info@DataScienceInTamil.com](mailto:info@DataScienceInTamil.com).
- All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content / code example. definitions, please use these websites for further reading:

- Book : Python Notes For Professionals
- <https://docs.python.org>
- <https://www.w3schools.com>
- <https://www.geeksforgeeks.org>
- <https://docs.python.org>
- <https://www.askpython.com>
- <https://docs.python.org>
- <https://www.programiz.com>

## 007 TOPIC: PYTHON LOOPS

For loop, while, break, continue

### What to cover today

1. **What is loop**
2. **Finite loop**
3. **Infinite loop**
4. **“Nested” loops**
5. **Nested Loops – how to use “break”**
6. **Loop with increment**
7. **'Break' in Loops**
8. **'Continue' in Loops**

- 9. A break statement inside a function cannot be used**
- 10. For loops**
- 11. For loops in list**
- 12. For loop with 'range'**
- 13. For loop with 'enumerate'**
- 14. Loops with an "else" clause**
- 15. else and continue inside the for loop**
- 16. The Pass Statement**
- 17. The "half loop" do-while**
- 18. enumerate and for loop**
- 19. Iterating different portion of a list with different step size**
- 20. Iterate over sub-list**

TOPIC : LOOPS

As one of the most basic functions in programming, loops are an important piece to nearly every programming language. Loops enable developers **to set certain portions of their code to repeat through a number of times** which are referred to as iterations. This topic covers using multiple types of loops and applications of loops in Python.

## Infinite loop

```
i = 0
while i < 7:
    print(i)
```

output

infinite

-----

## Loop with increment

(increment the initial value for **every iteration** of loop)

```
marks = 0
while marks < 35:
    print(marks)
    # marks = marks + 1
```

```
# marks = marks + 5
# # marks = marks + 10
```

*output*

*Another code*

```
for item in (3,5,7,9, 6):
    print(item)
    if item == 10:
        break
```

**note: since none of the item is equalent to 10, the exectuin never goes to break statement**

-----

## Break and Continue in Loops

When a **break** statement executes inside a loop, control flow "breaks" out of the loop **immediately**:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
```



```
        break
    i+=1
```

output

0

1

2

3

4

=====

Note: it prints all items, as the condition never met

It process the order in which seq of the tuple

```
for i in (3,5,7,9, 6):
    print(i)
    if i == 1:
        break
```

output

3

5

7

9

6

=====

```
for i in ("Kiran", 3,5,7,9, 6, "Linda", "Aswathy", " ", False, "Sudha"):
    print((i))
    if i == " ":
        break
```

=====

```
marks = 35
for name in ("Lavender", "Muthu", "TamilSelvan", "Muthu", "Siva", ): # list, set, tuple, dict
    print(name)
    if name == "TamilSelvan" and marks >= 35:
        print(name, " mark is : ", marks)
        break
```

-----

a=0

b=1

**while** a<7 **and** b>8 : *# since b becomes FALSE, this while loop returns FALSE, so the execution does not go the next line*

```
    print("a Value" , a , " b Value ",b)
    a=a+1
    b=b+2
```

=====

The loop conditional will not be evaluated after the **break** statement is executed. Note that **break** statements are **only allowed inside loops**, syntactically.

**A **break** statement inside a function cannot be used to terminate loops that called that function.**

Phycharm itself does not allow the “break” statement inside the fn – see below the example

```
def test():
    print("Canada")
    break # break can not be used inside the fn to stop the fn execution, it
is an error
for i in range (5):
    test() # ie we call the fn inside the loop, so the fn can not be broken
using break statement
```

-----

**break** statements can also be used inside **for** loops, the other looping construct provided by Python:

**break** statements can also be used inside **for** loops, the other looping construct provided by Python:

```
for i in (0, 1, 2, 3, 4):  
    print(i)  
    if i == 2:  
        break
```

Executing this loop now prints:

```
0  
1  
2
```

Note that 3 and 4 are not printed since the loop has ended.

If a loop has an **else** clause, it does not execute when the loop is terminated through a **break** statement.

```
for val in range(5):  
    print(val)  
    if val == 3:  
        break  
    else:  
        print("This Else Statement will bot be executed  
after the 3rd execution")
```

**output**

0

This Else Statement will bot be executed after the 3rd execution

1

This Else Statement will bot be executed after the 3rd execution

2

This Else Statement will bot be executed after the 3rd execution

3

=====

## Continue statement

It skips the **current iteration** but never comes out / terminate the loop

A **continue** statement will skip to the next iteration of the loop bypassing the rest of the current block but continuing the loop. As with **break**, **continue** can only appear inside loops:

```
lst = [2, 4, 6, 8, 10]
for item in lst:
    if item == 6:
        continue
    print(item)
```

output

```
2
4
8
10
```

```
=====
lst = [2, 4, 6, 8, 10]
for item in lst:
    if item == 2:
        break
    print(item)
```

no output ..why ? discuss and find the reason

=====

```
for i in (0,1,2,3,4):  
    if i == 2 or i == 4:  
        continue  
    print(i)
```

output

0  
1  
3

Note that 2 and 4 aren't printed, this is because **continue** goes to the next iteration instead of continuing on to **print(i)** when `i == 2` or `i == 4`.

-----

```
lst = [2,4,6,8,10]  
for item in lst:  
    if item == 2 and item == 4:  
        continue  
    print(item)
```

output

run the pgm to see the result and discuss

```
-----  
lst = [2,4,6,8,10]  
for item in lst:  
    if item == 2 or item == 4:  
        continue  
    print(item)
```

Output

6

8

10

Note: why 2 and 4 are not displayed? discuss

=====

## Nested Loops – how to use “break”

**break** and **continue** only **operate on a single level of loop**. The following example will only break out of the inner **for** loop, not the outer **while** loop:

```
while True:  
    for i in range(1, 5):
```



```
print(i)
if i == 2:
    break # Will only break the inner loop!
```

=====

```
while True:
    for i in range(1,5):
        print(i)
        if i == 2:
            break # it breaks the for loop
    break # # it breaks the while
```

output

1  
2

-----

Home work starts here

(Home work)

<https://cscircles.cemc.uwaterloo.ca/visualize#mode=display>

```
while True:
    for i in range(1,5):
        print(i)
        if i == 2:
```

**break**  
**break**

-----

Below print 5 times the 1 and 2 (Home work)

Use the below visualization

Check if there is any other viualiztion tools

<https://cscircles.cemc.uwaterloo.ca/visualize#mode=display>

```
a = 0
while a < 5:
    for i in range(1,5):
        print(i, a)
        if i == 2:
            break # Will only break out of the inner loop!
    a = a+1
```

ouput

```
1 0
2 0
1 1
2 1
1 2
2 2
```

```
1 3
2 3
1 4
2 4
```

how the result arrived ? discuss

-----

```
while True:
    for i in range(1,5):
        if i == 2:
            continue      # # continue skip that particular iteration
        print(i)
```

NOTE: it prints 134 continuously

Note: execute the pgm to see the result

In the above pg, it is a nested loops. Break and continue will break => only one level of loop. ie...the inner 'for' loop will be broken.

But the outer While loop will keep execute

To break all the loop together , use the below nested in inside a fn with 'return' – see below

-----

```
def m() :  
    while True:  
        for i in range(1, 5):  
            print(i)  
            if i == 2:  
                return
```

m()

output

1

2

---

```
def m() :  
    while True:  
        for i in range(1, 5):  
            if i == 2:  
                return  
            print(i)
```

m()

Note: **because of 'return' , the above will come out from inner 'for' loop and outer 'while' loop**

=====

**Another way of break the outerloop// do some more research**

```
while True:  
    for i in range(1,5):  
        print(i)  
        if i ==2:  
            break  
    break
```

-----

**Below is Home work / use the visualization**

**<https://cscircles.cemc.uwaterloo.ca/visualize#mode=display>**

**Any time the pgm sees return, it GETS OUT of the fn**

```
def m():  
    while True:  
        for i in range(1,5):
```

```
    if i == 2:  
        print(i)  
    print(i)  
    return
```

m()  
output

1

2

2

3

4

=====

Python doesn't have the ability to break out of multiple levels of loop at once -- if this behavior is desired, refactoring one or more loops into a function and replacing **break** with **return** may be the way to go.

=====

Use **return** from within a function as a **break**

The **return** statement **exits from a function**, without executing the code that comes after it.

**If you have a loop inside a function**, using **return** from inside that loop is equivalent to having a **break** as the rest of the code of the loop is not executed (note that any code after the loop is not executed either):

=====

Home work / visual tool

```
def breakLoop():  
    for i in range(1,5):  
        if i == 2:  
            return (i) #this returns 2  
        print(i)  
    return ""
```

```
a = breakLoop()  
print(a, "it comes from return")
```

output

1

2

Note: # if we don't assign this fn to a variable and print, the output 2 will not be available

---

```
def breakLoop():  
    for i in range(1, 5):  
        if i == 2:  
            return  
        print(i)  
    return (5)
```

```
a = breakLoop()  
print(a)
```

Output:

1

None

=====

Above pgm can be redefined as below

```
def breakLoop():  
    for i in range(1, 5):
```



```
        if i == 3:
            return (i) # this 'return' works only if the i
becomes 3
        print(i) # only 1 and 2 will print here
    return ()
```

```
a = breakLoop()
```

```
print(a)
```

output

1

2

3 # this 3 comes from return statement

=====

If you have nested loops, the **return** statement will break all loops:

```
def break_all():
    for j in range(1, 5):
        for i in range(1, 4):
            if i * j == 6:
                return (i)
        print(i * j)
```

```
a = break_all()  
print(a)
```

output

```
1 (1x1# 1 from outer loop, 1 from inner loop  
2 # (1x2) 1 from outer loop, 2 from inner loop  
3 # (1 x3) 1 from inner loop 3 from inner loop  
2 (2x1) 2 from outer loop, 1 from inner loop  
4 # (2x2) 2 from outer loop, 2 from inner loop
```

```
3 # 3 comes from return statement
```

```
-----  
1 # 1*1  
2 # 1*2  
3 # 1*3  
4 # 1*4  
2 # 2*1  
4 # 2*2
```

# return because  $2*3 = 6$ , the remaining iterations of both loops are not executed

Execute the code by the use of visual representation from  
<http://www.pythontutor.com/visualize.html#mode=display>

=====

For loops

**for** loops iterate over a collection of items, such as **list** or **dict**, and run a block of code with each element from the collection.

```
for i in [0, 1, 2, 3, 4]:  
    print(i)
```

The above **for** loop iterates over a list of numbers.

Each iteration sets the value of **i** to the next element of the list. So first it will be 0, then 1, then 2, etc. The output will be as follow:

```
0  
1  
2  
3  
4
```

=====

**range** is a function that returns a series / range of numbers under an iterable form, thus it can be used in **for** loops:

```
for i in range(5):  
    print(i)
```

gives the exact same result as the first **for** loop. Note that 5 is not printed as the range here is the first five numbers counting from 0.

### Iterable objects and iterators

**for** loop can iterate on any iterable object which is an object which defines a `__getitem__` or a `__iter__` function. The `__iter__` function returns an iterator, which is an object with a `next` function that is used to access the next element of the iterable.

```
a = li.__iter__()  
for i in a:  
    print(i)  
print(a)  
print(list(a))
```

Home work ends here

## Iterating over lists

To iterate through a list you can use **for**:

```
for x in ['one', 'two', 'three', 'four']:  
    print(x)
```

This will print out the elements of the list:

```
one  
two  
three  
four
```

The `range` function generates numbers which are also often used in a for loop.

```
for x in range(1, 6):  
    print(x)
```

The result will be a special [range sequence type](#) in python  $\geq 3$  and a list in python  $\leq 2$ . Both can be looped through using the for loop.

```
1  
2  
3  
4  
5
```

## Enumerate ()

If you want to loop through both the elements of a list and have an **index for the elements** as well, you can use Python's `enumerate` function:

```
lst = ['one', 'two', 'three', 'four']
```

```
for item in enumerate(lst):  
    # print (index, '::', item)  
    print (item)
```

output

(0, 'one')  
(1, 'two')  
(2, 'three')  
(3, 'four')

Note: **enumerate** will generate tuples, which are unpacked into index (an integer) and item (the actual value from the list). See the output above

-----

## Loops with an "else" clause

The **for** and **while** compound statements (loops) can optionally have an **else** clause (in practice, this usage is fairly rare).

1. The **else** clause only executes after a **for** loop terminates by iterating to completion, (ie all the iteration must be completed with out any interruption)
2. after a **while** loop terminates by its loop conditional expression **becoming false**

```
for i in range(3):  
    print (i)  
else:  
    print ("After the completion of the for loop")
```



output

0

1

2

After the completion of the for loop

-----

Now the for loop has some interruption (ie break) , so the else Class will not execute- see below

```
for i in range(5):  
    print (i)  
    if i== 3:  
        print("HI")  
        break  
else:  
    print("After the completion of the for loop")
```

output

0

1

2

3

=====

What happens if give else and continue inside the for loop

```
for i in range(5):  
    print (i)  
    if i== 3:  
        print("Hi")  
        continue  
else:  
    print("After the completion of the for loop")
```

output

```
0  
1  
2  
3  
Hi  
4  
After the completion of the for loop  
from Sean (from DSIT)
```

-----

```
while False: # else clause WILL execute  
    pass
```

```
else:
```

```
print("after LOOP the conditional expression becomes false, this ELSE will print")
print(10+20)
```

-----

```
while True: # else clause WILL NOT execute, 'cos it is an indefinite loop
    pass
```

```
else:
    print("after LOOP the conditional expression becomes false, this ELSE will print")
    print(10+20)
```

-----

```
i = 0
while i < 5:
    print(i)
    i+=1
else:
    print("after LOOP the conditional expression
becomes false, this will print")
```

output

0  
1  
2

3

4

after the conditional expression becomes false, this will print

The **else** clause does not execute if the loop terminates some other way (through a **break** statement or by raising an exception):

```
for i in range(2):  
    print(i)  
    if i == 1:  
        break  
else:  
    print('done')
```

output:

```
0  
1
```

Most other programming languages lack this optional **else** clause of loops. The use of the keyword **else** in particular is often considered confusing.

The original concept for such a clause dates back to Donald Knuth and the meaning of the **else** keyword becomes clear if we rewrite a loop in terms of **if** statements and **goto** statements from earlier days before structured programming or from a lower-level assembly language.

Note: in the **for** loop, if “break” executes, the “**else**” will not execute, it comes out of the pgm.

If the break statement **DOES not execute**, it WILL go the else statement  
“Else” in generally at the last statement

Melcose note: Teach the below code on isinstance(), once the 'class' and object are taught

```
li = [ "Muthu", "Nathan" , "Melcose"]  
for item in li:  
    if isinstance(item, int):  
        print("This is number", item)  
        break  
    else:  
        print("This is string," , item)  
else:  
    print("Only if list contains all string")
```

output

This is string, Muthu

This is string, Nathan

This is string, Melcose

Only if list contains all string

-----

```
li = [ "Muthu", "Nathan" , 7, "Melcose"]  
for item in li:  
    if isinstance(item, int):  
        print("This is number", item)  
        break  
    else:  
        print("This is string," , item)  
else:  
    print("Only if list contains all string")
```

output

This is string, Muthu

This is string, Nathan

This is number 7

=====

```
i = 0
```

```
if type(i) is not int:  
    print("Not int type")
```

```
else:  
    print("Yes it is int type")
```

output

Yes it is int type

=====

```
lst = [3, "DS", 5, 'A', 89.4, 8, 5]
```

```
for item in lst:  
    if type(item) is not int:  
        print(item)
```

## output

DS

A

89.4

=====

```
lst = [3, 5, 8, 5, 4]
for item in lst:
    if type(item) is not int:
        print(item)
        break
else:
    print("No exception")
```

output

No exception

-----

```
lst = [3, 5, 8, "A", 5, 4]
for item in lst:
    if type(item) is not int:
        print(item)
        break
    else:
        print(item)
```



```
else:  
    print("No exception")
```

output

3

5

8

A

=====

## The Pass Statement

`pass` is a null statement for when a statement is required by Python syntax (such as within the body of a `for` or `while` loop), but no action is required or desired by the programmer. This can be **useful as a placeholder** for code that is yet to be written.

```
for x in range(10):
```

```
pass #we don't want to do anything, or are not ready to do anything here, so we'll pass
```

In this example, nothing will happen. The **for** loop will complete without error, but no commands or code will be actioned. **pass** allows us to run our code successfully without having all commands and action fully implemented.

Similarly, **pass** can be used in **while** loops, as well as in selections and function definitions etc.

```
while x == y:  
    pass
```

```
while True:  
    pass
```

```
-----
```

```
for item in range(5):  
    pass
```

```
=====
```

Cover the below topic once the dict / mapping is taught

## The "half loop" do-while

Unlike other languages, Python doesn't have a do-until or a do-while construct (this will allow code to be **executed once before the condition is tested**). However, you can combine a **while True** with a **break** to achieve the same purpose

```
a = 10
while True:
    a = a-1
    print(a)
    if (a < 7):
        break
    print("done")
```

**output**

9

done

8

done

7

done

6

=====

Teach the below once the collection data type is taught (keep it pending)

## Looping and Unpacking

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

```
for item in collection:
```

```
    i1 = (item[0])
```

```
    i2 = (item[1])
```

```
    i3 = (item[2])
```

```
    print(i1)
```

**output**

a

x

1

-----  
Same code as above..but diff logic ..see below

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1',  
'2', '3')]  
for item in collection:  
    i1 = (item[0])  
    i2 = (item[1])  
    i3 = (item[2])  
  
    print(i1,i2,i3)
```

output

a b c

x y z

1 2 3

-----  
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]

# print(collection[1])

for item in collection:

print(item)

print(item[0])

print(item[1])

print(item[2])

output

```
('a', 'b', 'c')
```

a

b

c

```
('x', 'y', 'z')
```

x

y

z

```
('1', '2', '3')
```

1

2

3

-----

```
collection = [('Lara', 'CBE', '10'), ('Jon',  
'Chennai', '2'), ('Murugan', 'Kerala', '30')]
```

```
for item in collection:
```

```
    i1 = (item[0])
```

```
        # i2 = (item[1])
```

```
# i3 = (item[2])
```

```
print(i1)
```

**output**

Lara

Jon

Murugan

=====

---

If you want to loop over a list of tuples for example:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

instead of doing something like this:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

or something like this:

```
for item in collection:
```

---

[GoalKicker.com](http://GoalKicker.com) – Python® Notes for Professionals

---

```
i1, i2, i3 = item
# logic
```

You can simply do this:

```
for i1, i2, i3 in collection:
    # logic
```



This will also **work for most types of iterables**, not just tuples.

=====

## Iterating different portion of a list with different step size

Suppose you have a long list of elements and you are only interested in **every other element of the list**. Perhaps you only want to examine the **first or last elements**, or a **specific range of entries** in your list. Python has strong indexing built-in capabilities. Here are some examples of how to achieve these scenarios.

Here's a simple list that will be used throughout the examples:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

## Iteration over the whole list

To iterate over each element in the list, a **for** loop like below can be used:

```
for s in lst:  
    print s[:1] # print the first letter
```

The **for** loop assigns *s* for each element of *lst*. This will print:

```
a  
b  
c  
d  
e
```

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']  
for item in lst:
```

```
print(item[0])  
print(item[0:3])
```

output

a

alp

b

bra

c

cha

d

del

e

ech

=====

## enumerate

Often you need both the **element** and the **index** of that **element**. The **enumerate** keyword performs that task.

```
lst = ['alpha', 'bravo', 'charlie', 'delta',  
       'echo']  
for item in enumerate(lst):  
    print(item)
```

### output

```
(0, 'alpha')  
(1, 'bravo')  
(2, 'charlie')  
(3, 'delta')  
(4, 'echo')
```

-----

### Another code

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']  
for item, index in enumerate(lst):  
    print("item is ", item, "index is ", index)
```

=====

```
for idx, s in enumerate(lst):  
    print("%s has an index of %d" % (s, idx))
```

The index `idx` will start with zero and increment for each iteration, while the `s` will contain the element being processed. The previous snippet will output:

```
alpha has an index of 0  
bravo has an index of 1  
charlie has an index of 2  
delta has an index of 3  
echo has an index of 4
```

=====

## Iterate over sub-list (Teach this once list is taught)

If we want to iterate over a range (remembering that Python uses zero-based indexing), use the `range` keyword.

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']  
for i in range(len(lst)):  
    pass  
print(lst[2:4])
```

**output**

```
['charlie', 'delta']
```

```
=====
```

Another logic to iterate over sublist

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']  
for i in range(len(lst)):  
    print(i, lst[i])
```

```
print(lst.index('delta'))
```

output

```
0 alpha  
1 bravo  
2 charlie  
3 delta  
4 echo  
3
```

```
=====
```

The list may also be sliced. The following slice notation goes from element at index 1 to the end with a step of 2. The two **for** loops give the same result.

```
for s in lst[1::2]:  
    print(s)  
  
for i in range(1, len(lst), 2):  
    print(lst[i])
```

The above snippet outputs:

```
bravo  
delta
```

## While Loop

A **while** loop will cause the loop statements to be executed until the loop condition is falsey. The following code will execute the loop statements a total of 4 times.

```
i = 0
while i < 4:
    #loop statements
    i = i + 1
```

While the above loop can easily be translated into a more elegant **for** loop, **while** loops are useful for checking if some condition has been met. The following loop will continue to execute until `myObject` is ready.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

**while** loops can also run without a condition by using numbers (complex or real) or **True**:

```
=====
count = 5
while count:
    count = count - 1
    print(count)
```

output



4

3

2

1

0

-----

**while** loops can also run without a condition by using numbers (complex or real) or **True**:

```
while 5.8j:  
    print(" 11")
```

output: run infinite

-----

=====

```
while 5:  
    print(" 11")
```

output: run infinite

-----

If the condition is always true the while loop will run forever (infinite loop) **if it is not terminated by a break or return** statement or an exception

```
while True:  
    print("infinite")  
    break
```

output

infinite # executed only once and then encounter with 'break' statement

-----

Use 'return' to break the infinite loop in the fn

```
def fn():  
    while True:  
        print("infinite")  
        return
```

fn()

output

infinite

-----

**while** loops can also run without a condition by using numbers (complex or real) or **True**:

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:      # You can also replace complex_num with any number, True or a value of any
    type                # Prints 1j forever
    print(complex_num)
```

If the condition is always true the while loop will run forever (infinite loop) if it is not terminated by a break or return statement or an exception.

If the condition is always true , while loop will run forever (infinite loop)- if it is not terminated by a **break** or **return (statement)** or an **exception**

For example:

```
while loop_condition():  
    ...  
    if break_condition():  
        break  
    ...
```

is equivalent to:

```
# pseudocode  
  
<<start>>:  
if loop_condition():  
    ...  
    if break_condition():  
        goto <<end>>  
    ...  
goto <<start>>
```

---

GoalKicker.com – Python® Notes for Professionals

```
<<end>>:
```

These remain equivalent if we attach an **else** clause to each of them.

For example:

