Chapter 16: Loops

Parameter	Details
boolean expression	expression that can be evaluated in a boolean context, e.g. `x < 10`
variable	variable name for the current element from the `iterable`
iterable	anything that implements iterations

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 loops** which are referred to as iterations. This topic covers using multiple types of loops and applications of loops in Python.

Section 16.1: 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 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

==========

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.**

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

Phycharm itself does not all the “break” statement in side the fn

```
def test():
```

```
 print("Canada")
 break # this shows error in Pycharm
for i in range (5):
 test() # ie we call the fn in side 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:

```
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

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
========
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.

========

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 out of the inner  loop!
```

```
Below print 5 times the 1 and 2
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
```

```
- while True:
 for i in range(1,5):
 if i == 2:
 continue # Will only break out of the  inner loop!
```

```
 print(i)
```

NOTE: it prints 1345 continuously

Note: execute the pgm to see the result

In the above pg, it is a nested loops. Break and continue will breaks=>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

---

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

=======

Interview qn

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):

=====

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

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 i in range(1,5):
 for j 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 bythe use of visual representation from
http://www.pythontutor.com/visualize.html#mode=display

======

Section 16.2: 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 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))
```

Section 16.3: Iterating over lists

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 >=3 and a list in python <=2. Both can be looped through using the for loop.

```
1
2
3
4
5
```

Enumerate () Sankar

If you want to loop though 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 ----------

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

for index, item in enumerate(lst):
 print('({} :: {})' .format(index, item))
output
 (0 :: one)
 (1 :: two)
 (2 :: three)
 (3 :: four)
=====
```

Iterate over a list with value manipulation using map and lambda, i.e. apply lambda function on each element in the list:

```
x = map(lambda e: e.upper(),['one', 'two', 'three',
 'four'])
```

```
print(x) # from python 2x
print(list(x)) #use this (this is from python 3x
output
<map object at 0x0000015D3AF53B08>
 ['ONE', 'TWO', 'THREE', 'FOUR']
========
```

Section 16.4: 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).

The else clause only executes after a for loop terminates by iterating to completion, or after a while loop terminates by its 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
=====
i = 0
while i < 5:
 print(i)
 i+=1
else:
 print("after the conditional expression becomes
false, this will print")
output
0
1
2
3
4
```

after the conditional expression becomes false, this will print

---

```
counter = 5
while counter < 10:
 print(counter)
 counter += 1
 if True:
 print(' it will execute when if condition True')
 else:
 print("this will execute when if condition fails")
```

output

```
5
 it will execute when if condition True
6
 it will execute when if condition True
7
 it will execute when if condition True
8
 it will execute when if condition True
9
 it will execute when if condition True
```

```
 it will execute when if condition True
```

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

li = [**"SBI"**, **"Canara"** , **"IOB"**]

```
for item in li:
 if isinstance(item, int): # Mel, show the class how to use instance()
print("This is number", item)
 break
 else:
 print("This is string,", item)
else:
 print("Only if list contains all string")
```

output

This is string, SBI

This is string, Canara

This is string, IOB

Only if list contains all string

```
li = [ "SBI", "Canara" , 7, "IOB"]
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, SBI

This is string,

Canara This is

number 7

========

A `for` loop with an `else` clause can be understood the same way. Conceptually, there is a loop condition that remains True as long as the iterable object or sequence still has some remaining elements.

Why would one use this strange construct?

The main use case for the `for...else` construct is a concise implementation of search as for instance:

```
a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")
```

To make the `else` in this construct less confusing one can think of it as "*if not break*" or "*if not found*".

Some discussions on this can be found in [Python-ideas] Summary of for...else threads, Why does python use 'else' after for and while loops? , and Else Clauses on Loop Statements

```
i = 0
if type(i) is not int:
 print("Yes it is int type")
else:
 print("Not int type")
```

output

Not 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
```

```
lst = [3,"DS",5,'A', 89.4, 8,5, True]
for item in lst:
 if type(item) is not int and type(item) is not bool:
print(item)
```

output

```
DS
A
89.4
```

==========

Section 16.5: 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
============
```

Section 16.6: Iterating over dictionaries

Section 16.7: 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
=========
```

Section 16.8: 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', 'Numpy', 80), ('Gomathi', 'Pandas', 98), ('Murugan', 'SciKit-Learn',
60)] for element in collection:
 i1 = (element[0])
 i2 = (element[1])
 i3 = (element[2])

 # print(i1)
```

```
# print(i3)
```

output

```
80
98
60
==========
```

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

==========

Section 16.9: 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
========
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')
```

==========

```
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

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
=========
```

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


```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

This would output:

```
lst at 2 contains charlie
lst at 3 contains delta
```

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
```

Section 16.10: 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`:

```
class anObject:
 pass

 def isNotReady(self):
 print("AAA")

 def tryToGetReady(self):
 print("BBB")
```

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

output

AAA

The above is equal to
# while None:
# myObject.tryToGetReady()
```

========

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

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

output: run infinite

---

count =5

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

output

4

3

2

1

0

==========

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

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 infintite 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
    print(complex_num)   # Prints 1j forever
```

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