Loops

All Versions

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.

Examples

Break and Continue in Loops

break statement

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:
        print("Breaking from loop")
        break
    i += 1
```

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.

Executing the following prints every digit until number 4 when the break statement is met and the loop stops:

```
0
1
2
3
4
Breaking from loop
```

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.

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:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

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

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):
        if i == 2:
            break    # Will only break out of the inner loop!
```

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

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)
```

will output:

```
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
```

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.

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
```

Generated numbers using range is also gives a list as an output.

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

Result will not be in a list.

```
1
2
3
4
5
```

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:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):
    print(index, '::', item)
```

enumerate will generate tuples, which are unpacked into index (an integer) and item (the actual value from the list). The above loop will print

```
(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)
```

Output:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

NB: in Python 3.x map returns an iterator instead of a list so you in case you need a list you have to cast the result print(list(x)) (see http://stackoverflow.com/documentation/python/809/incompatibilities-between-python-2-and-python-3/8186/map) in http://stackoverflow.com/documentation/python/809/incompatibilities-between-python-2-and-python-3).

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('done')

i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

output:

```
 0
 1
 2
 done
```

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.

For example:

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

is equivalent to:

```
# pseudocode

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

<<end>>:
```

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

For example:

```
while loop_condition():
    ...
    if break_condition():
        break
    ...
else:
    print('done')
```

is equivalent to:

```
# pseudocode

<<start>>:
if loop_condition():
    ...
    if break_condition():
        goto <<end>>
    ...
    goto <<start>>
else:
    print('done')

<<end>>:
```

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.

Iterating over dictionaries

Considering the following dictionary:

```
 d = {"a": 1, "b": 2, "c": 3}
```

To iterate through its keys, you can use:

```
for key in d:
    print(key)
```

Output:

```
"a"
"b"
"c"
```

This is equivalent to:

```
for key in d.keys():
    print(key)
```

or in Python 2:

```
for key in d.iterkeys():
    print(key)
```

To iterate through its values, use:

```
for value in d.values():
    print(value)
```

Output:

```
1
2
3
```

To iterate through its keys and values, use:

```
for key, value in d.items():
    print(key, "::", value)
```

Output:

```
a :: 1
b :: 2
c :: 3
```

Note that in Python 2, `.keys()` , `.values()` and `.items()` return a `list` object. If you simply need to iterate trough the result, you can use the equivalent `.iterkeys()` , `.itervalues()` and `.iteritems()` .

The difference between `.keys()` and `.iterkeys()` , `.values()` and `.itervalues()` , `.items()` and `.iteritems()` is that the `iter*` methods are generators. Thus, the elements within the dictionary are yielded one by one as they are evaluated. When a `list` object is returned, all of the elements are packed into a list and then returned for further evaluation.

> Note also that in Python 3, Order of items printed in the above manner does not follow any order.

Improvements requested:

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
```

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

```
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.

```
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
```

Indexing and slicing is a topic of its own.

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.')
```

This will print:

```
9
8
7
6
Done.
```

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 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 :

```
import cmath

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

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

Looping and Unpacking

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:
    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.

Syntax

```
while <boolean expression>:
```

```
for <variable> in <iterable>:
```

```
for <variable> in range(<number>):
```

```
for <variable> in range(<start_number>, <end_number>):
```

```
for i, <variable> in enumerate(<iterable>): # with index i
```

```
for <variable1>, <variable2> in zip(<iterable1>, <iterable2>):
```

Parameters

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

Remarks