Python "while" Loops (Indefinite Iteration)

Table of Contents

Mastering While Loops

Iteration means executing the same block of code over and over, potentially many times. A programming structure that implements iteration is called a loop.

In programming, there are two types of iteration, indefinite and definite:

- With indefinite iteration, the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some condition is met.
- With definite iteration, the number of times the designated block will be executed is specified explicitly at the time the loop starts.

In this tutorial, you'll:

- Learn about the `while` loop, the Python control structure used for indefinite iteration
- See how to break out of a loop or loop iteration prematurely
- Explore infinite loops

When you're finished, you should have a good grasp of how to use indefinite iteration in Python.

The `while` Loop

Let’s see how Python’s `while` statement is used to construct loops. We’ll start simple and embellish as we go.

The format of a rudimentary `while` loop is shown below:

```
while <expr>:
    <statement(s)>
```

`<statement(s)>` represents the block to be repeatedly executed, often referred to as the body of the loop. This is denoted with indentation, just as in an `if` statement.

Remember: All control structures in Python use indentation to define blocks. See the discussion on grouping statements in the previous tutorial to review.

The controlling expression, `<expr>`, typically involves one or more variables that are initialized prior to starting the loop and then modified somewhere in the loop body.

When a `while` loop is encountered, `<expr>` is first evaluated in Boolean context. If it is true, the loop body is executed. Then `<expr>` is checked again, and if still true, the body is executed again. This continues until `<expr>` becomes false, at which point program execution proceeds to the first statement beyond the loop body.

Consider this loop:

>>>

```
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
...
4
3
2
1
0
```

Here’s what’s happening in this example:

- `n` is initially `5`. The expression in the `while` statement header on line 2 is `n > 0`, which is true, so the loop body executes. Inside the loop body on line 3, `n` is decremented by `1` to `4`, and then printed.
- When the body of the loop has finished, program execution returns to the top of the loop at line 2, and the expression is evaluated again. It is still true, so the body executes again, and `3` is printed.
- This continues until `n` becomes `0`. At that point, when the expression is tested, it is false, and the loop terminates. Execution would resume at the first statement following the loop body, but there isn't one in this case.

Note that the controlling expression of the `while` loop is tested first before anything else happens. If it's false to start with, the loop body will never be executed at all:

```
>>>
>>> n = 0
>>> while n > 0:
...     n -= 1
...     print(n)
...
```

In the example above, when the loop is encountered, `n` is `0`. The controlling expression `n > 0` is already false, so the loop body never executes.

Here's another `while` loop involving a list, rather than a numeric comparison:

```
>>>
>>> a = ['foo', 'bar', 'baz']
>>> while a:
...     print(a.pop(-1))
...
baz
bar
foo
```

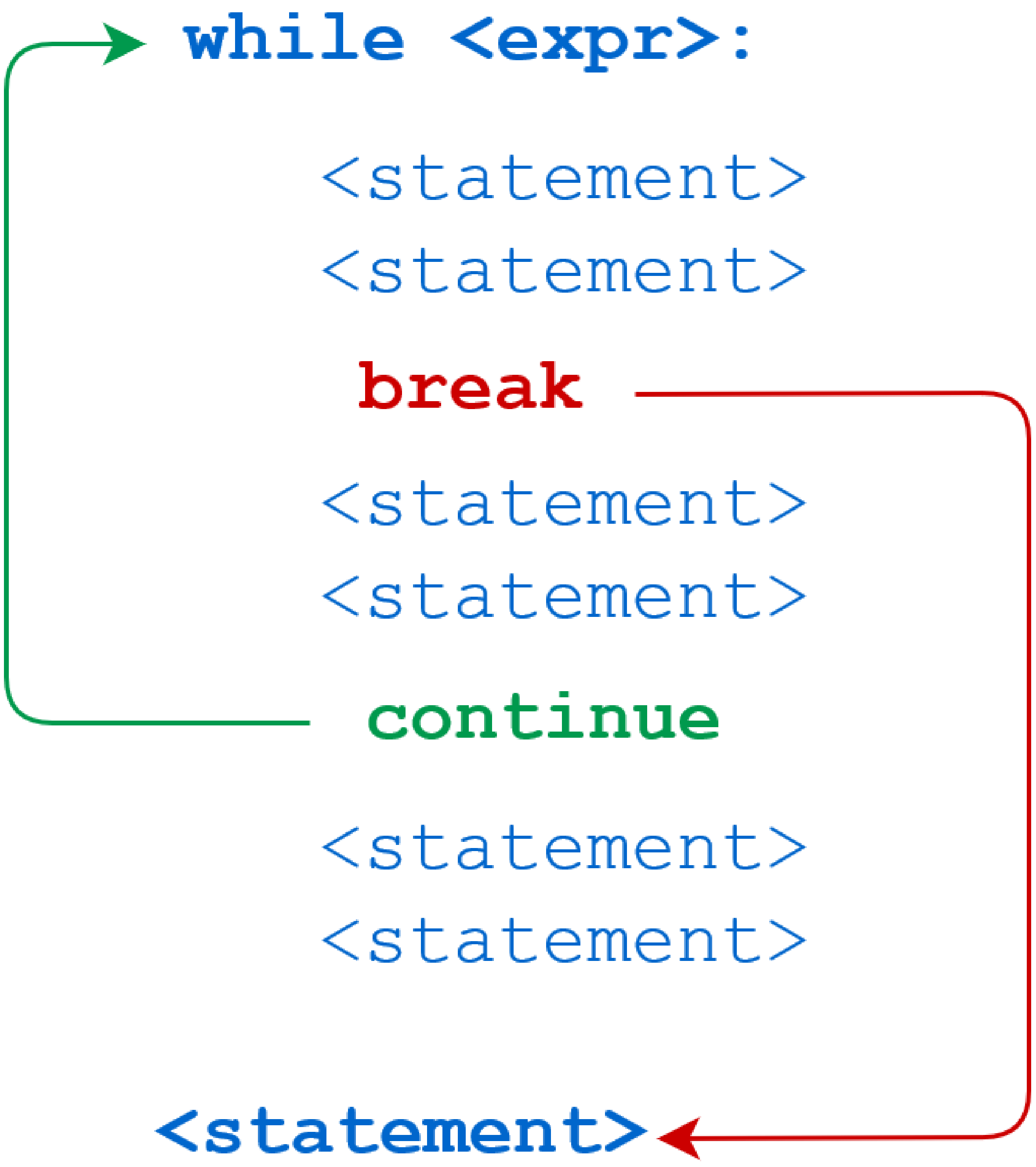
When a list is evaluated in Boolean context, it is truthy if it has elements in it and falsy if it is empty. In this example, `a` is true as long as it has elements in it. Once all the items have been removed with the `.pop()` method and the list is empty, `a` is false, and the loop terminates.

The Python `break` and `continue` Statements

In each example you have seen so far, the entire body of the `while` loop is executed on each iteration. Python provides two keywords that terminate a loop iteration prematurely:

- The Python `break` statement immediately terminates a loop entirely. Program execution proceeds to the first statement following the loop body.
- The Python `continue` statement immediately terminates the current loop iteration. Execution jumps to the top of the loop, and the controlling expression is re-evaluated to determine whether the loop will execute again or terminate.

The distinction between `break` and `continue` is demonstrated in the following diagram:



break and continue

Here’s a script file called `break.py` that demonstrates the `break` statement:

```
n = 5
while n > 0:
    n -= 1
    if n == 2:
```

```
5         break
6     print(n)
7 print('Loop ended.')
```

Running `break.py` from a command-line interpreter produces the following output:

```
C:\Users\john\Documents>python break.py
4
3
Loop ended.
```

When `n` becomes `2`, the `break` statement is executed. The loop is terminated completely, and program execution jumps to the `print()` statement on line 7.

The next script, `continue.py`, is identical except for a `continue` statement in place of the `break`:

```
1 n = 5
2 while n > 0:
3     n -= 1
4     if n == 2:
5         continue
6     print(n)
7 print('Loop ended.')
```

The output of `continue.py` looks like this:

```
C:\Users\john\Documents>python continue.py
4
3
1
0
Loop ended.
```

This time, when `n` is `2`, the `continue` statement causes termination of that iteration. Thus, `2` isn't printed. Execution returns to the top of the loop, the condition is re-evaluated, and it is still true. The loop resumes, terminating when `n` becomes `0`, as previously.

The `else` Clause

Python allows an optional `else` clause at the end of a `while` loop. This is a unique feature of Python, not found in most other programming languages. The syntax is shown below:

```
while <expr>:
    <statement(s)>
else:
    <additional_statement(s)>
```

The `<additional_statement(s)>` specified in the `else` clause will be executed when the `while` loop terminates.

About now, you may be thinking, “How is that useful?” You could accomplish the same thing by putting those statements immediately after the `while` loop, without the `else`:

```
while <expr>:
    <statement(s)>
<additional_statement(s)>
```

What’s the difference?

In the latter case, without the `else` clause, `<additional_statement(s)>` will be executed after the `while` loop terminates, no matter what.

When `<additional_statement(s)>` are placed in an `else` clause, they will be executed only if the loop terminates “by exhaustion”—that is, if the loop iterates until the controlling condition becomes false. If the loop is exited by a `break` statement, the `else` clause won’t be executed.

Consider the following example:

```
>>>
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
... else:
...     print('Loop done.')
...
4
3
2
1
0
Loop done.
```

In this case, the loop repeated until the condition was exhausted: `n` became `0`, so `n > 0` became false. Because the loop lived out its natural life, so to speak, the `else` clause was executed. Now observe the difference here:

```
>>>
```

```
>>> n = 5
>>> while n > 0:
...     n -= 1
...     print(n)
...     if n == 2:
...         break
... else:
...     print('Loop done.')
...
4
3
2
```

This loop is terminated prematurely with `break`, so the `else` clause isn't executed.

It may seem as if the meaning of the word `else` doesn't quite fit the `while` loop as well as it does the `if` statement. Guido van Rossum, the creator of Python, has actually said that, if he had it to do over again, he'd leave the `while` loop's `else` clause out of the language.

One of the following interpretations might help to make it more intuitive:

- Think of the header of the loop (`while n > 0`) as an `if` statement (`if n > 0`) that gets executed over and over, with the `else` clause finally being executed when the condition becomes false.
- Think of `else` as though it were `nobreak`, in that the block that follows gets executed if there wasn't a `break`.

If you don't find either of these interpretations helpful, then feel free to ignore them.

When might an `else` clause on a `while` loop be useful? One common situation is if you are searching a list for a specific item. You can use `break` to exit the loop if the item is found, and the `else` clause can contain code that is meant to be executed if the item isn't found:

```
>>>
>>> a = ['foo', 'bar', 'baz', 'qux']
>>> s = 'corge'

>>> i = 0
>>> while i < len(a):
...     if a[i] == s:
...         # Processing for item found
...         break
...     i += 1
... else:
...     # Processing for item not found
...     print(s, 'not found in list.')
...
corge not found in list.
```

Note: The code shown above is useful to illustrate the concept, but you'd actually be very unlikely to search a list that way.

First of all, lists are usually processed with definite iteration, not a `while` loop. Definite iteration is covered in the next tutorial in this series.

Secondly, Python provides built-in ways to search for an item in a list. You can use the `in` operator:

```
>>>
>>> if s in a:
...     print(s, 'found in list.')
... else:
...     print(s, 'not found in list.')
...
corge not found in list.
```

The `list.index()` method would also work. This method raises a `ValueError` exception if the item isn't found in the list, so you need to understand exception handling to use it. In Python, you use a `try` statement to handle an exception. An example is given below:

```
>>>
>>> try:
...     print(a.index('corge'))
... except ValueError:
...     print(s, 'not found in list.')
```

```
...
corge not found in list.
```

You will learn about exception handling later in this series.

An `else` clause with a `while` loop is a bit of an oddity, not often seen. But don't shy away from it if you find a situation in which you feel it adds clarity to your code!

Infinite Loops

Suppose you write a `while` loop that theoretically never ends. Sounds weird, right?

Consider this example:

```
>>>
>>> while True:
...     print('foo')
...
foo
foo
foo
  .
  .
  .
foo
foo
foo
KeyboardInterrupt
Traceback (most recent call last):
  File "<pyshell#2>", line 2, in <module>
    print('foo')
```

This code was terminated by `Ctrl+C`, which generates an interrupt from the keyboard. Otherwise, it would have gone on unendingly. Many `foo` output lines have been removed and replaced by the vertical ellipsis in the output shown.

Clearly, `True` will never be false, or we're all in very big trouble. Thus, `while True:` initiates an infinite loop that will theoretically run forever.

Maybe that doesn't sound like something you'd want to do, but this pattern is actually quite common. For example, you might write code for a service that starts up and runs forever accepting service requests. "Forever" in this context means until you shut it down, or until the heat death of the universe, whichever comes first.

More prosaically, remember that loops can be broken out of with the `break` statement. It may be more straightforward to terminate a loop based on conditions recognized within the loop body, rather than on a condition evaluated at the top.

Here's another variant of the loop shown above that successively removes items from a list using `.pop()` until it is empty:

```
>>>
>>> a = ['foo', 'bar', 'baz']
>>> while True:
...     if not a:
...         break
...     print(a.pop(-1))
...
baz
bar
foo
```

When `a` becomes empty, `not a` becomes true, and the `break` statement exits the loop.

You can also specify multiple `break` statements in a loop:

```
while True:
    if <expr1>:  # One condition for loop termination
        break
    ...
    if <expr2>:  # Another termination condition
        break
    ...
    if <expr3>:  # Yet another
        break
```

In cases like this, where there are multiple reasons to end the loop, it is often cleaner to `break` out from several different locations, rather than try to specify all the termination conditions in the loop header.

Infinite loops can be very useful. Just remember that you must ensure the loop gets broken out of at some point, so it doesn't truly become infinite.

Nested `while` Loops

In general, Python control structures can be nested within one another. For example, `if`/`elif`/`else` conditional statements can be nested:

```
if age < 18:
    if gender == 'M':
        print('son')
    else:
        print('daughter')
elif age >= 18 and age < 65:
    if gender == 'M':
        print('father')
    else:
        print('mother')
else:
    if gender == 'M':
        print('grandfather')
    else:
        print('grandmother')
```

Similarly, a `while` loop can be contained within another `while` loop, as shown here:

```
>>>
>>> a = ['foo', 'bar']
>>> while len(a):
...     print(a.pop(0))
...     b = ['baz', 'qux']
...     while len(b):
...         print('>', b.pop(0))
...
foo
> baz
> qux
bar
> baz
> qux
```

A `break` or `continue` statement found within nested loops applies to the nearest enclosing loop:

```
while <expr1>:
    statement
    statement

    while <expr2>:
        statement
        statement
        break  # Applies to while <expr2>: loop

    break  # Applies to while <expr1>: loop
```

Additionally, `while` loops can be nested inside `if`/`elif`/`else` statements, and vice versa:

```
if <expr>:
    statement
    while <expr>:
        statement
        statement
else:
    while <expr>:
        statement
        statement
    statement
```

```
while <expr>:
    if <expr>:
        statement
    elif <expr>:
        statement
    else:
        statement
    if <expr>:
        statement
```

In fact, all the Python control structures can be intermingled with one another to whatever extent you need. That is as it should be. Imagine how frustrating it would be if there were unexpected restrictions like “A `while` loop can’t be contained within an `if` statement” or “`while` loops can only be nested inside one another at most four deep.” You’d have a very difficult time remembering them all.

Seemingly arbitrary numeric or logical limitations are considered a sign of poor program language design. Happily, you won’t find many in Python.

One-Line `while` Loops

As with an `if` statement, a `while` loop can be specified on one line. If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons (`;`):

```
>>>
>>> n = 5
>>> while n > 0: n -= 1; print(n)
```

```
4
3
2
1
0
```

This only works with simple statements though. You can’t combine two compound statements into one line. Thus, you can specify a `while` loop all on one line as above, and you write an `if` statement on one line:

```
>>>
>>> if True: print('foo')
```

```
foo
```

But you can’t do this:

```
>>>
>>> while n > 0: n -= 1; if True: print('foo')
SyntaxError: invalid syntax
```

Remember that PEP 8 discourages multiple statements on one line. So you probably shouldn’t be doing any of this very often anyhow.

Conclusion

In this tutorial, you learned about indefinite iteration using the Python `while` loop. You’re now able to:

- Construct basic and complex `while` loops
- Interrupt loop execution with `break` and `continue`
- Use the `else` clause with a `while` loop
- Deal with infinite loops

You should now have a good grasp of how to execute a piece of code repetitively.