Previous: Python Boolean and Conditional Programming

Next: Python Function ›

Home » Introduction to Python » Python For Loop and While Loop

Python For Loop and While Loop

June 5, 2022

We learned how we can change the flow of our program with the conditional statements if and else. Another way to control the flow is by using a Python for-loop or a Python while-loop. Loops, in essence, allow you to repeat a piece of code.

Table of Contents [hide]

1 Python For-loop
2 Python for-loops and lists
3 Python While-loop
4 Infinite loops
5 More types of loops

Beginners Python Course (2023) **Modules, Packages, And Virtual Environments (2023)**

Python For-loop

There are two ways to create a loop in Python. Let's first look at Python's for-loop. A for-loop iterates over the individual elements of the object you feed it. If that sounds difficult, an example will hopefully clarify this:

```
>>> for letter in 'Hello':
...     print(letter)
...
H
e
l
l
o
```

We stumbled upon two concepts here that need an explanation: iterability and objects.

Iterable

An iterable is an object in Python that can return its members one at a time.

As you can see in the example code, a string of text is iterable. Most of Python's data types are iterable in some way or another. If you want to know all the nitty-gritty details, head over to the page about iterators. If you're new to programming, I suggest you keep reading here and save iterators for later.

The next thing we need to tackle: objects. This is a big subject, and it has its own chapter in this tutorial: Python classes and objects. You don't need to learn about it now to understand Python for-loops. It's enough to know that everything in Python is an object, and objects have a certain type and certain properties. Being iterable is one of those properties.

So we know that a for-loop can loop over iterable objects. By returning its members one by one, making it available in a variable (in the above example it's the variable `letter`), you can loop over each element of an iterable with a for-statement.

The general template for a for-loop in Python is:

```
for <variable> in <iterable>:
    ... do something with variable
```

On each iteration, an element from `iterable` is assigned to `variable`. This variable exists and can be used only inside the loop. Once there is nothing more left, the loop stops and the program continues with the next lines of code.

Python for-loops and lists

This is the ideal time to look at a new data type: lists. A Python list can contain zero or more objects. It's a frequently used data type in Python programming. In other programming languages, there often is no such thing as a list. Most languages do offer arrays, but arrays can only contain one type of data.

Python has arrays too, but we won't discuss them in this course. In short: arrays can only contain one type of data (like numbers) but they are more efficient at storing that data. Lists, on the other hand, can store a mix of all types of data and are extremely flexible, at the cost of some performance. For a thorough explanation of lists, you can read my article on Python lists. If you're a beginner, I recommend to keep on reading here though.

Lists, just like strings, are iterable; hence they work very well with a for-loop:

```
>>> mylist = [1, 'a', 'Hello']
>>> for item in mylist:
...     print(item)
...
1
a
Hello
```

Let's try that again, but this time in an interactive crumb so you can play around with it:

A couple of things to note:

- we create lists with block quotes.

- Its contents are objects of whatever type you like, separated by commas, and they don't need to be of the same type.
- We can access the individual elements of a list manually too.

A list can contain all the types we've seen so far: numbers, strings, booleans, and even other lists. Indeed, you can create a list of lists if you want! As you can see in the interactive example, we can also access individual elements of a list. Remember that in computer science, we start counting from 0. So `mylist[0]` gives us the first element, and `mylist[1]` the second, etcetera.

Here are a couple of things you can try in a REPL or in the interactive code example above:

```
>>> mylist = [1, 2, 'Hello', ['a', 'b'] ]
>>> mylist[0]
1
>>> mylist[0] + mylist[1]
3
>>> mylist[2]
'Hello'
>>> mylist[3][0]
'a'
```

From the last example, you can see how to access nested lists. An extensive tutorial on Python lists can be found later on in this Python tutorial.

Python While-loop

While the for-loop in Python is a bit hard to understand, because of new concepts like iterability and objects, the while loop is actually much simpler! Its template looks like this:

```
while <expression evaluates to True>:
    do something
```

You should read this as: "while this expression evaluates to

`True` , keep doing the stuff below".

Let's take a look at an actual example:

```
>>> i = 1
>>> while i <= 4:
```

```
...     print(i)
...     i = i + 1
...
1
2
3
4
```

We see an expression that follows the `while` statement: `i <= 4`. As long as this expression evaluates to `True`, the block inside of the while-loop executes repeatedly.

In the example above, we start with `i = 1`. In the first iteration of the loop, we print `i` and increase it by one. This keeps happening as long as `i` is smaller than or equal to 4. The output of the print statement confirms that this loop runs four times.

Here's an interactive example to experiment with yourself:

The Python while loop

Infinite loops

Sometimes you want your software to keep running. In such cases, an infinite loop can be of help. Let's create such an infinite loop in Python:

```
>>> while True:
...     print("Help I'm stuck in a loop!")
...
```

Why is this an infinite loop? Remember that while takes an expression, and keeps repeating the code as long as that expression evaluates to `True`. Since the very simple expression '`True`' is always `True`, this loop never stops.

Getting out of an Infinite loop

With while loops, it's easy to make a mistake and find yourself caught in an infinite while-loop. This means the expression never evaluates to False for some reason. It happens to the best of us. You can get out of this situation by pressing **control + c** at the same time. Since we know how to intentionally create infinite loops, let's artificially create such a situation. Hit **control + c** to stop the following loop:

```
>>> while True:
```

```
...     print("Help I'm stuck in a loop!")
...
```

The output will look like this:

```
Help I'm stuck in a loop!
Help I'm stuck in a loop!
Help I'm stuck in a loop!
Help I'm stuck in a loop!
Help I'm stuck in a loop!
Help I'm stuck in a loop!^C
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

If you look closely, you see the characters `^C` right before the error, meaning `control + c` was pressed at that point. This key combination will get you out of most situations where your program runs indefinitely, so it's good to remember it!

Infinite loops are less common in for-loops because most iterable objects will at some point run out of elements to iterate over. However, if you ever find yourself in an infinite for-loop, you can use this same trick to get out of it.

More types of loops

Later on in the tutorial, you will also learn about Python list comprehensions. A list comprehension is a powerful construct in Python that we can use to create a list based on an existing list. If you're new to programming, you should learn about other essentials first, though and I suggest you continue with the following section.

Get certified with our courses

Learn Python properly through small, easy-to-digest lessons, progress tracking, quizzes to test your knowledge, and practice sessions. Each course will earn you a downloadable course certificate.

‹ Previous: Python Boolean and Conditional Programming

Next: Python Function