

Python "for" Loops (Definite Iteration)

Table of Contents

- A Survey of Definite Iteration in Programming
 - Numeric Range Loop
 - Three-Expression Loop
 - Collection-Based or Iterator-Based Loop
- The Python for Loop
 - Iterables
 - Iterators
- The Guts of the Python for Loop
- Iterating Through a Dictionary
- The range() Function
- Altering for Loop Behavior
 - The break and continue Statements
 - The else Clause
- Conclusion

This tutorial will show you how to perform definite iteration with a Python `for` loop.

In the previous tutorial in this introductory series, you learned the following:

- Repetitive execution of the same block of code over and over is referred to as iteration.
- There are two types of iteration:
 - Definite iteration, in which the number of repetitions is specified explicitly in advance
 - Indefinite iteration, in which the code block executes until some condition is met
- In Python, indefinite iteration is performed with a `while` loop.

Here's what you'll cover in this tutorial:

- You'll start with a comparison of some different paradigms used by programming languages to implement definite iteration.
- Then you will learn about iterables and iterators, two concepts that form the basis of definite iteration in Python.
- Finally, you'll tie it all together and learn about Python's `for` loops.

A Survey of Definite Iteration in Programming

Definite iteration loops are frequently referred to as `for` loops because `for` is the keyword that is used to introduce them in nearly all programming languages, including Python.

Historically, programming languages have offered a few assorted flavors of `for` loop. These are briefly described in the following sections.

Numeric Range Loop

The most basic `for` loop is a simple numeric range statement with start and end values. The exact format varies depending on the language but typically looks something like this:

```
for i = 1 to 10
    <loop body>
```

Here, the body of the loop is executed ten times. The `variable` `i` assumes the value `1` on the first iteration, `2` on the second, and so on. This sort of `for` loop is used in the languages BASIC, Algol, and Pascal.

Three-Expression Loop

Another form of `for` loop popularized by the C programming language contains three parts:

- An initialization
- An expression specifying an ending condition
- An action to be performed at the end of each iteration.

This type of loop has the following form:

```
for (i = 1; i <= 10; i++)
    <loop body>
```

Technical Note: In the C programming language, `i++` increments the variable `i`. It is roughly equivalent to `i += 1` in Python.

This loop is interpreted as follows:

- Initialize `i` to `1`.
- Continue looping as long as `i <= 10`.
- Increment `i` by `1` after each loop iteration.

Three-expression `for` loops are popular because the expressions specified for the three parts can be nearly anything, so this has quite a bit more flexibility than the simpler numeric range form shown above. These `for` loops are also featured in the C++, Java, PHP, and Perl languages.

Collection-Based or Iterator-Based Loop

This type of loop iterates over a collection of objects, rather than specifying numeric values or conditions:

```
for i in <collection>
    <loop body>
```

Each time through the loop, the variable `i` takes on the value of the next object in `<collection>`. This type of `for` loop is arguably the most generalized and abstract. Perl and PHP also support this type of loop, but it is introduced by the keyword `foreach` instead of `for`.

The Python `for` Loop

Of the loop types listed above, Python only implements the last: collection-based iteration. At first blush, that may seem like a raw deal, but rest assured that Python's implementation of definite iteration is so versatile that you won't end up feeling cheated!

Shortly, you'll dig into the guts of Python's `for` loop in detail. But for now, let's start with a quick prototype and example, just to get acquainted.

Python's `for` loop looks like this:

```
for <var> in <iterable>:
    <statement(s)>
```

`<iterable>` is a collection of objects—for example, a list or tuple. The `<statement(s)>` in the loop body are denoted by indentation, as with all Python control structures, and are executed once for each item in `<iterable>`. The loop variable `<var>` takes on the value of the next element in `<iterable>` each time through the loop.

Here is a representative example:

```
>>>
>>> a = ['foo', 'bar', 'baz']
>>> for i in a:
...     print(i)
...
foo
bar
baz
```

In this example, `<iterable>` is the list `a`, and `<var>` is the variable `i`. Each time through the loop, `i` takes on a successive item in `a`, so `print()` displays the values '`foo`', '`bar`', and '`baz`', respectively. A `for` loop like this is the Pythonic way to process the items in an iterable.

But what exactly is an iterable? Before examining `for` loops further, it will be beneficial to delve more deeply into what iterables are in Python.

Iterables

In Python, iterable means an object can be used in iteration. The term is used as:

- An adjective: An object may be described as iterable.
- A noun: An object may be characterized as an iterable.

If an object is iterable, it can be passed to the built-in Python function `iter()`, which returns something called an iterator. Yes, the terminology gets a bit repetitive. Hang in there. It all works out in the end.

Each of the objects in the following example is an iterable and returns some type of iterator when passed to `iter()`:

```
>>>
>>> iter('foobar')                                     # String
<str_iterator object at 0x036E2750>

>>> iter(['foo', 'bar', 'baz'])                      # List
<list_iterator object at 0x036E27D0>

>>> iter(('foo', 'bar', 'baz'))                      # Tuple
<tuple_iterator object at 0x036E27F0>

>>> iter({'foo', 'bar', 'baz'})                       # Set
<set_iterator object at 0x036DEA08>

>>> iter({'foo': 1, 'bar': 2, 'baz': 3})            # Dict
<dict_keyiterator object at 0x036DD990>
```

These object types, on the other hand, aren't iterable:

```
>>>
>>> iter(42)                                         # Integer
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    iter(42)
TypeError: 'int' object is not iterable

>>> iter(3.1)                                         # Float
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
```

```
    iter(3.1)
TypeError: 'float' object is not iterable

>>> iter(len)                                     # Built-in function
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    iter(len)
TypeError: 'builtin_function_or_method' object is not iterable
```

All the data types you have encountered so far that are collection or container types are iterable. These include the `string`, `list`, `tuple`, `dict`, `set`, and `frozenset` types.

But these are by no means the only types that you can iterate over. Many objects that are built into Python or defined in modules are designed to be iterable. For example, open files in Python are iterable. As you will see soon in the tutorial on file I/O, iterating over an open file object reads data from the file.

In fact, almost any object in Python can be made iterable. Even user-defined objects can be designed in such a way that they can be iterated over. (You will find out how that is done in the upcoming article on object-oriented programming.)

Iterators

Okay, now you know what it means for an object to be iterable, and you know how to use `iter()` to obtain an iterator from it. Once you've got an iterator, what can you do with it?

An iterator is essentially a value producer that yields successive values from its associated iterable object. The built-in function `next()` is used to obtain the next value from in iterator.

Here is an example using the same list as above:

```
>>>
>>> a = ['foo', 'bar', 'baz']

>>> itr = iter(a)
>>> itr
<list_iterator object at 0x031EFD10>

>>> next(itr)
'foo'
>>> next(itr)
'bar'
>>> next(itr)
'baz'
```

In this example, `a` is an iterable list and `itr` is the associated iterator, obtained with `iter()`. Each `next(itr)` call obtains the next value from `itr`.

Notice how an iterator retains its state internally. It knows which values have been obtained already, so when you call `next()`, it knows what value to return next.

What happens when the iterator runs out of values? Let's make one more `next()` call on the iterator above:

```
>>>
>>> next(itr)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    next(itr)
StopIteration
```

If all the values from an iterator have been returned already, a subsequent `next()` call raises a `StopIteration` exception. Any further attempts to obtain values from the iterator will fail.

You can only obtain values from an iterator in one direction. You can't go backward. There is no `prev()` function. But you can define two independent iterators on the same iterable object:

```
>>>
>>> a
['foo', 'bar', 'baz']

>>> itr1 = iter(a)
>>> itr2 = iter(a)

>>> next(itr1)
'foo'
>>> next(itr1)
'bar'
>>> next(itr1)
'baz'

>>> next(itr2)
'foo'
```

Even when iterator `itr1` is already at the end of the list, `itr2` is still at the beginning. Each iterator maintains its own internal state, independent of the other.

If you want to grab all the values from an iterator at once, you can use the built-in `list()` function. Among other possible uses, `list()` takes an iterator as its argument, and returns a list consisting of all the values that the iterator yielded:

```
>>>
>>> a = ['foo', 'bar', 'baz']
>>> itr = iter(a)
>>> list(itr)
['foo', 'bar', 'baz']
```

Similarly, the built-in `tuple()` and `set()` functions return a tuple and a set, respectively, from all the values an iterator yields:

```
>>>
>>> a = ['foo', 'bar', 'baz']

>>> itr = iter(a)
>>> tuple(itr)
('foo', 'bar', 'baz')

>>> itr = iter(a)
>>> set(itr)
{'baz', 'foo', 'bar'}
```

It isn't necessarily advised to make a habit of this. Part of the elegance of iterators is that they are "lazy." That means that when you create an iterator, it doesn't generate all the items it can yield just then. It waits until you ask for them with `next()`. Items are not created until they are requested.

When you use `list()`, `tuple()`, or the like, you are forcing the iterator to generate all its values at once, so they can all be returned. If the total number of objects the iterator returns is very large, that may take a long time.

In fact, it is possible to create an iterator in Python that returns an endless series of objects using generator functions and `itertools`. If you try to grab all the values at once from an endless iterator, the program will hang.

The Guts of the Python `for` Loop

You now have been introduced to all the concepts you need to fully understand how Python's `for` loop works. Before proceeding, let's review the relevant terms:

Term	Meaning
Iteration	The process of looping through the objects or items in a collection
Iterable	An object (or the adjective used to describe an object) that can be iterated over
Iterator	The object that produces successive items or values from its associated iterable
<code>iter()</code>	The built-in function used to obtain an iterator from an iterable

Now, consider again the simple `for` loop presented at the start of this tutorial:

```
>>>
>>> a = ['foo', 'bar', 'baz']
>>> for i in a:
...     print(i)
...
foo
bar
baz
```

This loop can be described entirely in terms of the concepts you have just learned about. To carry out the iteration this `for` loop describes, Python does the following:

- Calls `iter()` to obtain an iterator for `a`
- Calls `next()` repeatedly to obtain each item from the iterator in turn
- Terminates the loop when `next()` raises the `StopIteration` exception

The loop body is executed once for each item `next()` returns, with loop variable `i` set to the given item for each iteration.

This sequence of events is summarized in the following diagram:

Iterable

```
a = ['foo', 'bar', 'baz']
```

```
iter()
```

```
itr = iter(a)
```

Iterator

```
next(itr) → 'foo'
```

```
next(itr) → 'bar'
```

```
next(itr) → 'baz'
```

```
next(itr)
```

StopIteration

Schematic Diagram of a Python for Loop

Perhaps this seems like a lot of unnecessary monkey business, but the benefit is substantial. Python treats looping over all iterables in exactly this way, and in Python, iterables and iterators abound:

- Many built-in and library objects are iterable.
- There is a Standard Library module called `itertools` containing many functions that return iterables.
- User-defined objects created with Python's object-oriented capability can be made to be iterable.
- Python features a construct called a generator that allows you to create your own iterator in a simple, straightforward way.

You will discover more about all the above throughout this series. They can all be the target of a `for` loop, and the syntax is the same across the board. It's elegant in its simplicity and eminently versatile.

Iterating Through a Dictionary

You saw earlier that an iterator can be obtained from a dictionary with `iter()`, so you know dictionaries must be iterable. What happens when you loop through a dictionary? Let's see:

```
>>>
>>> d = {'foo': 1, 'bar': 2, 'baz': 3}
>>> for k in d:
...     print(k)
...
foo
bar
baz
```

As you can see, when a `for` loop iterates through a dictionary, the loop variable is assigned to the dictionary's keys.

To access the dictionary values within the loop, you can make a dictionary reference using the key as usual:

```
>>>
>>> for k in d:
...     print(d[k])
...
1
2
3
```

You can also iterate through a dictionary's values directly by using `.values()`:

```
>>>
>>> for v in d.values():
...     print(v)
...
1
2
3
```

In fact, you can iterate through both the keys and values of a dictionary simultaneously. That is because the loop variable of a `for` loop isn't limited to just a single variable. It can also be a tuple, in which case the assignments are made from the items in the iterable using packing and unpacking, just as with an assignment statement:

```
>>>
>>> i, j = (1, 2)
>>> print(i, j)
1 2

>>> for i, j in [(1, 2), (3, 4), (5, 6)]:
...     print(i, j)
...
1 2
3 4
5 6
```

As noted in the tutorial on Python [dictionaries](#), the dictionary method `.items()` effectively returns a list of key/value pairs as tuples:

```
>>>
>>> d = {'foo': 1, 'bar': 2, 'baz': 3}

>>> d.items()
dict_items([('foo', 1), ('bar', 2), ('baz', 3)])
```

Thus, the Pythonic way to iterate through a dictionary accessing both the keys and values looks like this:

```
>>>
>>> d = {'foo': 1, 'bar': 2, 'baz': 3}
>>> for k, v in d.items():
...     print('k =', k, ', v =', v)
...
k = foo , v = 1
k = bar , v = 2
k = baz , v = 3
```

The `range()` Function

In the first section of this tutorial, you saw a type of `for` loop called a [numeric range loop](#), in which starting and ending numeric values are specified. Although this form of `for` loop isn't directly built into Python, it is easily arrived at.

For example, if you wanted to iterate through the values from `0` to `4`, you could simply do this:

```
>>>
>>> for n in (0, 1, 2, 3, 4):
...     print(n)
...
0
1
2
3
4
```

This solution isn't too bad when there are just a few numbers. But if the number range were much larger, it would become tedious pretty quickly.

Happily, Python provides a better option—the built-in `range()` function, which returns an iterable that yields a sequence of integers.

`range(<end>)` returns an iterable that yields integers starting with `0`, up to but not including `<end>`:

```
>>>
>>> x = range(5)
>>> x
range(0, 5)
>>> type(x)
<class 'range'>
```

Note that `range()` returns an object of class `range`, not a list or tuple of the values. Because a `range` object is an iterable, you can obtain the values by iterating over them with a `for` loop:

```
>>>
>>> for n in x:
...     print(n)
...
0
1
2
3
4
```

You could also snag all the values at once with `list()` or `tuple()`. In a REPL session, that can be a convenient way to quickly display what the values are:

```
>>>  
>>> list(x)  
[0, 1, 2, 3, 4]  
  
>>> tuple(x)  
(0, 1, 2, 3, 4)
```

However, when `range()` is used in code that is part of a larger application, it is typically considered poor practice to use `list()` or `tuple()` in this way. Like iterators, `range` objects are lazy—the values in the specified range are not generated until they are requested. Using `list()` or `tuple()` on a `range` object forces all the values to be returned at once. This is rarely necessary, and if the list is long, it can waste time and memory.

`range(<begin>, <end>, <stride>)` returns an iterable that yields integers starting with `<begin>`, up to but not including `<end>`. If specified, `<stride>` indicates an amount to skip between values (analogous to the `stride` value used for string and list slicing):

```
>>>  
>>> list(range(5, 20, 3))  
[5, 8, 11, 14, 17]
```

If `<stride>` is omitted, it defaults to 1:

```
>>>  
>>> list(range(5, 10, 1))  
[5, 6, 7, 8, 9]  
>>> list(range(5, 10))  
[5, 6, 7, 8, 9]
```

All the parameters specified to `range()` must be integers, but any of them can be negative. Naturally, if `<begin>` is greater than `<end>`, `<stride>` must be negative (if you want any results):

```
>>>  
>>> list(range(-5, 5))  
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]  
  
>>> list(range(5, -5))  
[]
```

```
>>> list(range(5, -5, -1))
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Technical Note: Strictly speaking, `range()` isn't exactly a built-in function. It is implemented as a callable class that creates an immutable sequence type. But for practical purposes, it behaves like a built-in function.

For more information on `range()`, see the Real Python article [Python's `range\(\)` Function \(Guide\)](#).

Altering for Loop Behavior

You saw in the previous tutorial in this introductory series how execution of a `while` loop can be interrupted with `break` and `continue` statements and modified with an `else` clause. These capabilities are available with the `for` loop as well.

The `break` and `continue` Statements

`break` and `continue` work the same way with `for` loops as with `while` loops. `break` terminates the loop completely and proceeds to the first statement following the loop:

```
>>>
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     if 'b' in i:
...         break
...     print(i)
...
foo
```

`continue` terminates the current iteration and proceeds to the next iteration:

```
>>>
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     if 'b' in i:
...         continue
...     print(i)
...
foo
qux
```

The `else` Clause

A `for` loop can have an `else` clause as well. The interpretation is analogous to that of a `while` loop. The `else` clause will be executed if the loop terminates through exhaustion of the iterable:

```
>>>
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     print(i)
... else:
...     print('Done.') # Will execute
...
foo
bar
baz
qux
Done.
```

The `else` clause won't be executed if the list is broken out of with a `break` statement:

```
>>>
>>> for i in ['foo', 'bar', 'baz', 'qux']:
...     if i == 'bar':
...         break
...     print(i)
... else:
...     print('Done.') # Will not execute
...
foo
```

Conclusion

This tutorial presented the `for` loop, the workhorse of definite iteration in Python.

You also learned about the inner workings of iterables and iterators, two important object types that underlie definite iteration, but also figure prominently in a wide variety of other Python code.

In the next two tutorials in this introductory series, you will shift gears a little and explore how Python programs can interact with the user via input from the keyboard and output to the console.

