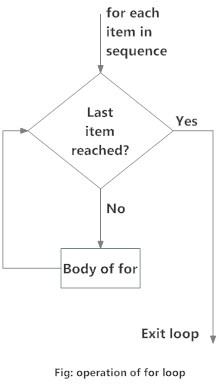
**Loops**

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

There may be a situation when you need to execute a block of code several number of times.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −





**Iteration:**

* 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.

**\*\*\*Definite iteration loops are frequently referred to as for loops**

**#Indefinite** iteration, in which the code block executes until some

condition is met

**\*\*\*In Python, indefinite iteration is performed with a while loop.**

**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.

**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 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>.

**The Python for Loop**

Python only implements the last: collection-based iteration.

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 = ['apple', 'orange', 'cherry']

>>> for i in a:

... print(i)

...

apple

orange

cherry

**In this example**

**1. <iterable> is the list a**

2. **<var> is the variable i**.

Each time through the loop, i takes on a successive item in a, so print() displays the values 'apple', 'orange', and 'cherry', respectively. A for loop like this is the Pythonic way to process the items in an iterable.

**Iterating Through a Dictionary**

>>> d = {'apple': 1, 'orange ': 2, 'cherry': 3}

>>> for k in d:

... print(k)

...

apple

orange

cherry

**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.

**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

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

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**

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:

>>>

>>> x=range(5)

>>> 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.

**Altering for Loop Behavior**

The break and continue Statements

**break terminates the loop completely and proceeds to the first statement following the loop:**

>>>

>>> for i in ['cherry', 'orange', 'apple', 'banana']:

... if 'a' in i:

... break

... print(i)

...

cherry

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

>>>

>>> for i in ['cherry', 'orange', 'apple', 'banana']:

... if 'a' in i:

... continue

... print(i)

...

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)

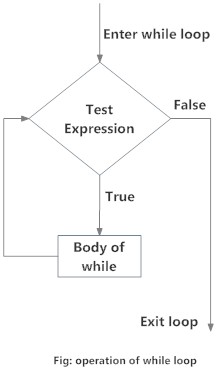
... else:

... print('Done.') # Will not execute

...

foo

**The while Loop**



Let’s see how Python’s while statement is used to construct loops.

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.

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](https://realpython.com/python-data-types/#boolean-type-boolean-context-and-truthiness). 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:

>>>

1 >>> n = 5

2 >>> while n > 0:

3 ... n -= 1

4 ... print(n)

5 ...

6 4

7 3

8 2

9 1

10 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())

...

baz

bar

foo

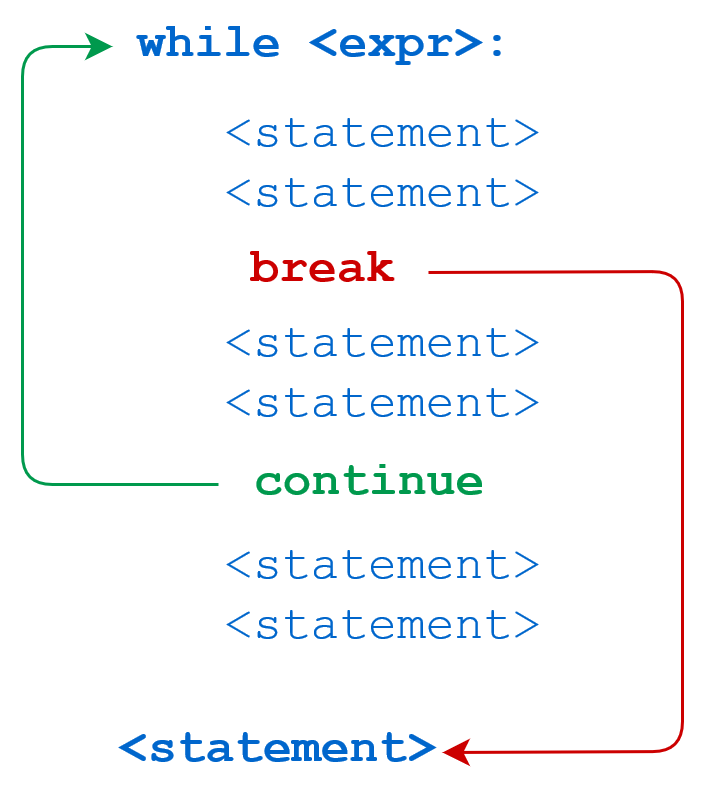
When a [list is evaluated in Boolean context](https://realpython.com/python-operators-expressions/#evaluation-of-non-boolean-values-in-boolean-context), it is truthy if it has elements in it and falsy if it is empty. In this example, a is true as long as it has elements in it. Once all the items have 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:

[](https://files.realpython.com/media/t.899f357dd948.png)

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

1 n = 5

2 while n > 0:

3 n -= 1

4 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](https://en.wikipedia.org/wiki/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.

**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())

...

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:

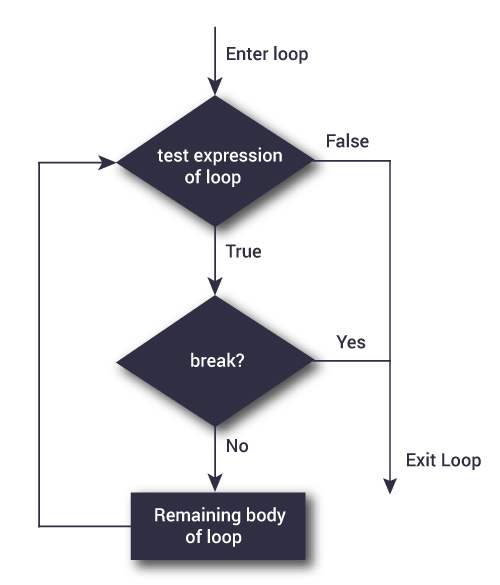
>>>

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

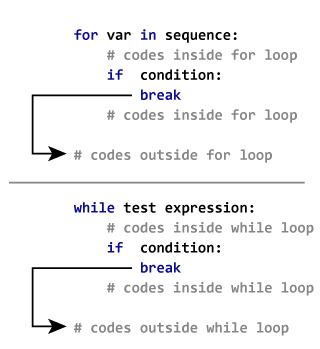
SyntaxError: invalid syntax

Remember that [PEP 8](https://www.python.org/dev/peps/pep-0008/#other-recommendations) discourages multiple statements on one line. So you probably shouldn’t be doing any of this very often anyhow.

### lowchart of break



The working of break statement in [for loop](https://www.programiz.com/python-programming/for-loop) and [while loop](https://www.programiz.com/python-programming/while-loop) is shown below.



**# Use of break statement inside loop**

**for val in "string":**

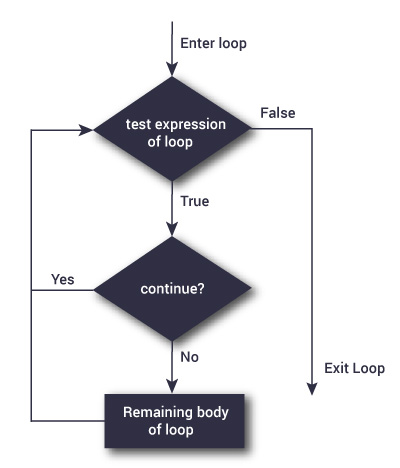
**if val == "i":**

**break**

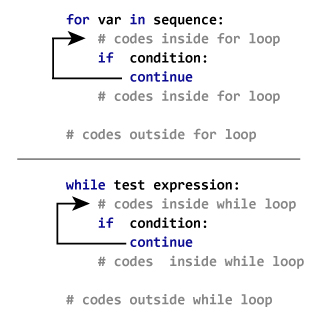
**print(val)**

**print("The end")**

**Continue**



The working of continue statement in for and while loop is shown below.



**# Program to show the use of continue statement inside loops**

**for val in "string":**

**if val == "i":**

**continue**

**print(val)**

**print("The end")**

**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](https://realpython.com/python-strings/), [list](https://realpython.com/python-lists-tuples/#python-lists), [tuple](https://realpython.com/python-lists-tuples/#python-tuples), [dict](https://realpython.com/python-dicts/), [set](https://realpython.com/python-sets/), and [frozenset](https://realpython.com/python-sets/" \l "frozen-sets) 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. (You will learn how to do this in upcoming tutorials on generator functions and itertools.) If you try to grab all the values at once from an endless iterator, the program will [hang](https://en.wikipedia.org/wiki/Hang_(computing)).

**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 |
| **iter()** | 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

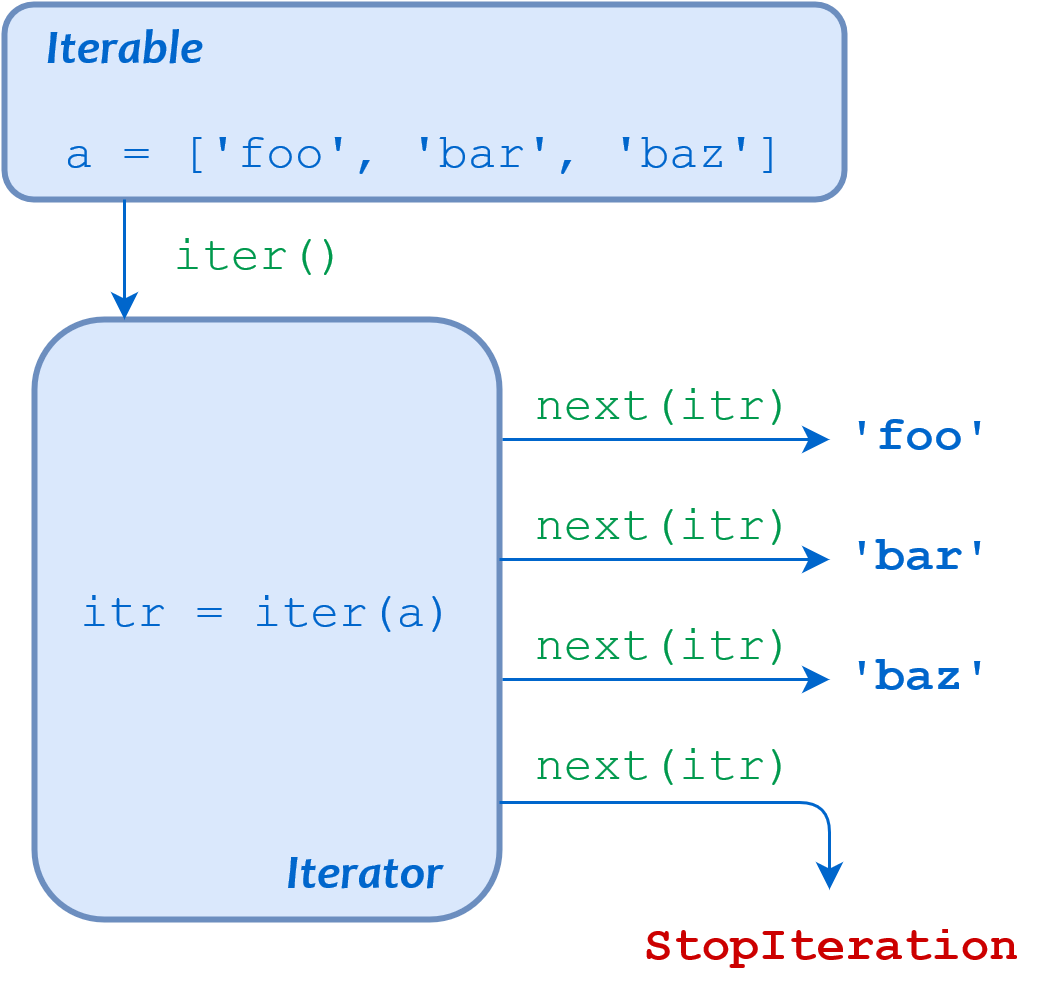
baz

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:

[](https://files.realpython.com/media/t.ba63222d63f5.png)

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.