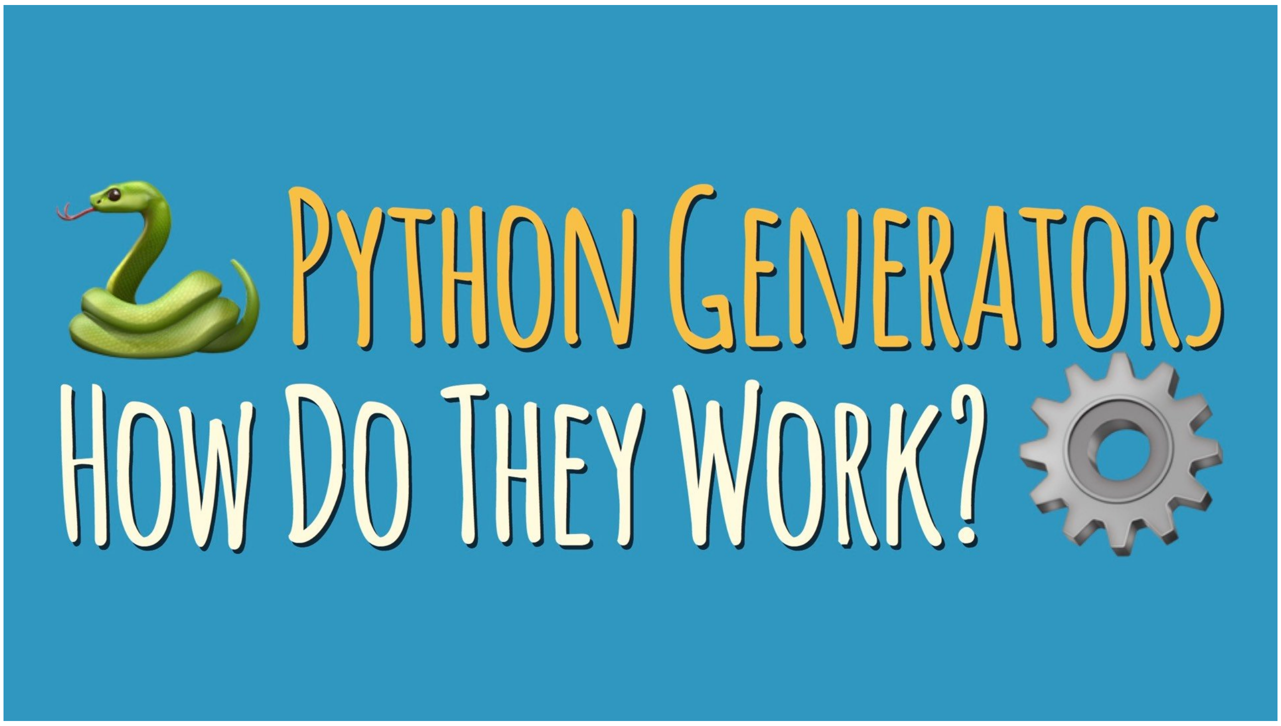
# What Are Python Generators?

dbader.org/blog/python-generators

By Dan Bader — Get free updates of new posts here.



If you've ever implemented a class-based iterator from scratch in Python, you know that this endeavour requires writing quite a bit of boilerplate code.

And yet, iterators are so useful in Python: They allow you to write pretty *for-in* loops and help you make your code more Pythonic and efficient.

As a (proud) "lazy" Python developer, I don't like tedious and repetitive work. And so, I often found myself wondering:

> If there only was a more convenient way to write these Python iterators in the first place…

Surprise, there is! Once again, Python helps us out with some syntactic sugar to make writing iterators easier.

**In this tutorial you'll see how to write Python iterators faster and with less code using *generators* and the `yield` keyword.**

Ready? Let's go!

## Python Generators 101 – The Basics

Let's start by looking again at the `Repeater` example that I previously used to introduce the idea of iterators. It implemented a class-based iterator cycling through an infinite sequence of values.

This is what the class looked like in its second (simplified) version:

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

If you're thinking, "that's quite a lot of code for such a simple iterator," you're absolutely right. Parts of this class seem rather formulaic, as if they would be written in exactly the same way from one class-based iterator to the next.

This is where Python's *generators* enter the scene. If I rewrite this iterator class as a generator, it looks like this:

```
def repeater(value):
    while True:
        yield value
```

**We just went from seven lines of code to three.**

Not bad, eh? As you can see, generators look like regular functions but instead of using the `return` statement, they use `yield` to pass data back to the caller.

Will this new generator implementation still work the same way as our class-based iterator did? Let's bust out the *for-in* loop test to find out:

```
>>> for x in repeater('Hi'):
...    print(x)
'Hi'
'Hi'
'Hi'
'Hi'
'Hi'
...
```

Yep! We're still looping through our greetings forever. This much shorter *generator* implementation seems to perform the same way that the `Repeater` class did.

*(Remember to hit Ctrl+C if you want out of the infinite loop in an interpreter session.)*

Now, how do these generators work? They look like normal functions, but their behavior is quite different. For starters, calling a generator function doesn't even run the function. It merely creates and returns a *generator object*:

```
>>> repeater('Hey')
<generator object repeater at 0x107bcdbf8>
```

The code in the generator function only executes when next() is called on the generator object:

```
>>> generator_obj = repeater('Hey')
>>> next(generator_obj)
'Hey'
```

If you read the code of the `repeater` function again, it looks like the `yield` keyword in there somehow stops this generator function in mid-execution and then resumes it at a later point in time:

```
def repeater(value):
    while True:
        yield value
```

And that's quite a fitting mental model for what happens here. You see, when a `return` statement is invoked inside a function, it permanently passes control back to the caller of the function. When a `yield` is invoked, it also passes control back to the caller of the function—but it only does so *temporarily*.

**Whereas a return statement disposes of a function's local state, a yield statement suspends the function and retains its local state.**

In practical terms, this means local variables and the execution state of the generator function are only stashed away temporarily and not thrown out completely.

Execution can be resumed at any time by calling `next()` on the generator:

```
>>> iterator = repeater('Hi')
>>> next(iterator)
'Hi'
>>> next(iterator)
'Hi'
>>> next(iterator)
'Hi'
```

This makes generators fully compatible with the iterator protocol. For this reason, I like to think of them primarily as syntactic sugar for implementing iterators.

You'll find that for most types of iterators, writing a generator function will be easier and more readable than defining a long-winded class-based iterator.

## Python Generators That Stop Generating

In this tutorial we started out by writing an *infinite* generator once again. By now you're probably wondering how to write a generator that stops producing values after a while, instead of going on and on forever.

Remember, in our class-based iterator we were able to signal the end of iteration by manually raising a StopIteration exception. Because generators are fully compatible with class-based iterators, that's still what happens behind the scenes.

Thankfully, as programmers we get to work with a nicer interface this time around. Generators stop generating values as soon as control flow returns from the generator function by any means other than a `yield` statement. This means you no longer have to worry about raising StopIteration at all!

Here's an example:

```
def repeat_three_times(value):
    yield value
    yield value
    yield value
```

Notice how this generator function doesn't include any kind of loop. In fact it's dead simple and only consists of three `yield` statements. If a `yield` temporarily suspends execution of the function and passes back a value to the caller, what will happen when we reach the end of this generator?

Let's find out:

```
>>> for x in repeat_three_times('Hey there'):
...     print(x)
'Hey there'
'Hey there'
'Hey there'
```

As you may have expected, this generator stopped producing new values after three iterations. We can assume that it did so by raising a `StopIteration` exception when execution reached the end of the function.

But to be sure, let's confirm that with another experiment:

```
>>> iterator = repeat_three_times('Hey there')
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
```

This iterator behaved just like we expected. As soon as we reach the end of the generator function, it keeps raising `StopIteration` to signal that it has no more values to provide.

Let's come back to another example from my Python iterators tutorials. The `BoundedIterator` class implemented an iterator that would only repeat a value a set number of times:

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

Why don't we try to re-implement this `BoundedRepeater` class as a generator function. Here's my first take on it:

```
def bounded_repeater(value, max_repeats):
    count = 0
    while True:
        if count >= max_repeats:
            return
        count += 1
        yield value
```

I intentionally made the `while` loop in this function a little unwieldy. I wanted to demonstrate how invoking a `return` statement from a generator causes iteration to stop with a `StopIteration` exception. We'll soon clean up and simplify this generator function some more, but first let's try out what we've got so far:

```
>>> for x in bounded_repeater('Hi', 4):
...     print(x)
'Hi'
'Hi'
'Hi'
'Hi'
```

Great! Now we have a generator that stops producing values after a configurable number of repetitions. It uses the `yield` statement to pass back values until it finally hits the `return` statement and iteration stops.

Like I promised you, we can further simplify this generator. We'll take advantage of the fact that Python adds an implicit `return None` statement to the end of every function. This is what our final implementation looks like:

```
def bounded_repeater(value, max_repeats):
    for i in range(max_repeats):
        yield value
```

Feel free to confirm that this simplified generator still works the same way. All things considered, we went from a 12-line iterator in the `BoundedRepeater` class to a three-line generator-based implementation providing the same functionality.

**That's a 75% reduction in the number of lines of code—not too shabby!**

Generator functions are a great feature in Python, and you shouldn't hesitate to use them in your own programs.

As you just saw, generators help you "abstract away" most of the boilerplate code otherwise needed when writing class-based iterators. Generators can make your life as a Pythonista much easier and allow you to write cleaner, shorter, and more maintainable iterators.

## Python Generators – A Quick Summary

- Generator functions are syntactic sugar for writing objects that support the iterator protocol. Generators abstract away much of the boilerplate code needed when writing class-based iterators.
- The `yield` statement allows you to temporarily suspend execution of a generator function and to pass back values from it.
- Generators start raising `StopIteration` exceptions after control flow leaves the generator function by any means other than a `yield` statement.