Python Iterators: A Step-By-Step Introduction

dbader.org/blog/python-iterators

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

I love how beautiful and clear Python's syntax is compared to many other programming languages.

Let's take the humble *for-in* loop, for example. It speaks for Python's beauty that you can read a Pythonic loop like this as if it was an English sentence:

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

But how do Python's elegant loop constructs work behind the scenes? How does the loop fetch individual elements from the object it is looping over? And how can you support the same programming style in your own Python objects?

You'll find the answer to these questions in Python's iterator protocol:

> Objects that support the `__iter__` and `__next__` dunder methods automatically work with *for-in* loops.

But let's take things step by step. Just like decorators, iterators and their related techniques can appear quite arcane and complicated on first glance. So we'll ease into it.

In this tutorial you'll see how to write several Python classes that support the iterator protocol. They'll serve as "non-magical" examples and test implementations you can build upon and deepen your understanding with.

We'll focus on the core mechanics of iterators in Python 3 first and leave out any unnecessary complications, so you can see clearly how iterators behave at the fundamental level.

I'll tie each example back to the *for-in* loop question we started out with. And at the end of this tutorial we'll go over some differences that exist between Python 2 and 3 when it comes to iterators.

Ready? Let's jump right in!

Python Iterators That Iterate Forever

We'll begin by writing a class that demonstrates the bare-bones iterator protocol in Python. The example I'm using here might look different from the examples you've seen in other iterator tutorials, but bear with me. I think doing it this way gives you a more applicable understanding of how iterators work in Python.

Over the next few paragraphs we're going to implement a class called `Repeater` that can be iterated over with a *for-in* loop, like so:

```
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

Like its name suggests, instances of this `Repeater` class will repeatedly return a single value when iterated over. So the above example code would print the string `Hello` to the console forever.

To start with the implementation we'll define and flesh out the `Repeater` class first:

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

    def __iter__(self):
        return RepeaterIterator(self)
```

On first inspection, `Repeater` looks like a bog-standard Python class. But notice how it also includes the `__iter__` dunder method.

What's the `RepeaterIterator` object we're creating and returning from `__iter__` ? It's a helper class we also need to define for our *for-in* iteration example to work:

```
class RepeaterIterator:
    def __init__(self, source):
        self.source = source

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

Again, `RepeaterIterator` looks like a straightforward Python class, but you might want to take note of the following two things:

1. In the `__init__` method we link each `RepeaterIterator` instance to the `Repeater` object that created it. That way we can hold on to the "source" object that's being iterated over.

2. In `RepeaterIterator.__next__`, we reach back into the "source" `Repeater` instance and return the value associated with it.

In this code example, `Repeater` and `RepeaterIterator` are working *together* to support Python's iterator protocol. The two dunder methods we defined, `__iter__` and `__next__`, are the key to making a Python object iterable.

We'll take a closer look at these two methods and how they work together after some hands-on experimentation with the code we've got so far.

Let's confirm that this two-class setup really made `Repeater` objects compatible with *for-in* loop iteration. To do that we'll first create an instance of `Repeater` that would return the string `'Hello'` indefinitely:

```
>>> repeater = Repeater('Hello')
```

And now we're going to try iterating over this repeater object with a *for-in* loop. What's going to happen when you run the following code snippet?

```
>>> for item in repeater:
...     print(item)
```

Right on! You'll see `'Hello'` printed to the screen...a lot. `Repeater` keeps on returning the same string value, and so, this loop will never complete. Our little program is doomed to print `'Hello'` to the console forever:

```
Hello
Hello
Hello
Hello
Hello
...
```

But congratulations—you just wrote a working iterator in Python and used it with a *for-in* loop. The loop may not terminate yet...but so far, so good!

Next up we'll tease this example apart to understand how the `__iter__` and `__next__`

methods work together to make a Python object iterable.

Pro tip: If you ran the last example inside a Python REPL session or from the terminal and you want to stop it, hit *Ctrl + C* a few times to break out of the infinite loop.

How do *for-in* loops work in Python?

At this point we've got our `Repeater` class that apparently supports the iterator protocol, and we just ran a *for-in* loop to prove it:

```
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

Now, what does this for-in loop really do behind the scenes? How does it communicate with the `repeater` object to fetch new elements from it?

To dispel some of that "magic" we can expand this loop into a slightly longer code snippet that gives the same result:

```
repeater = Repeater('Hello')
iterator = repeater.__iter__()
while True:
    item = iterator.__next__()
    print(item)
```

As you can see, the *for-in* was just syntactic sugar for a simple `while` loop:

- It first prepared the `repeater` object for iteration by calling its `__iter__` method. This returned the actual *iterator object*.
- After that, the loop repeatedly calls the iterator object's `__next__` method to retrieve values from it.

If you've ever worked with *database cursors*, this mental model will seem familiar: We first initialize the cursor and prepare it for reading, and then we can fetch data into local variables as needed from it, one element at a time.

Because there's never more than one element "in flight", this approach is highly memory-efficient. Our `Repeater` class provides an *infinite* sequence of elements and we can iterate over it just fine. Emulating the same with a Python `list` would be impossible—there's no way we could create a list with an infinite number of elements in the first place. This makes iterators a very powerful concept.

On more abstract terms, iterators provide a common interface that allows you to process every element of a container while being completely isolated from the container's internal structure.

Whether you're dealing with a list of elements, a dictionary, an infinite sequence like the one provided by our `Repeater` class, or another sequence type—all of that is just an

implementation detail. Every single one of these objects can be traversed in the same way by the power of iterators.

And as you've seen, there's nothing special about *for-in* loops in Python. If you peek behind the curtain, it all comes down to calling the right dunder methods at the right time.

In fact, you can manually "emulate" how the loop used the iterator protocol in a Python interpreter session:

```
>>> repeater = Repeater('Hello')
>>> iterator = iter(repeater)
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
...
```

This gives the same result: An infinite stream of hellos. Every time you call `next()` the iterator hands out the same greeting again.

By the way, I took the opportunity here to replace the calls to `__iter__` and `__next__` with calls to Python's built-in functions `iter()` and `next()`.

Internally these built-ins invoke the same dunder methods, but they make this code a little prettier and easier to read by providing a clean "facade" to the iterator protocol.

Python offers these facades for other functionality as well. For example, `len(x)` is a shortcut for calling `x.__len__`. Similarly, calling `iter(x)` invokes `x.__iter__` and calling `next(x)` invokes `x.__next__`.

Generally it's a good idea to use the built-in facade functions rather than directly accessing the dunder methods implementing a protocol. It just makes the code a little easier to read.

A Simpler Iterator Class

Up until now our iterator example consisted of two separate classes, `Repeater` and `RepeaterIterator`. They corresponded directly to the two phases used by Python's iterator protocol:

First setting up and retrieving the iterator object with an `iter()` call, and then repeatedly fetching values from it via `next()`.

Many times *both of these responsibilities* can be shouldered by a single class. Doing this allows you to reduce the amount of code necessary to write a class-based iterator.

I chose not to do this with the first example in this tutorial, because it mixes up the cleanliness of the mental model behind the iterator protocol. But now that you've seen how to write a class-based iterator the longer and more complicated way, let's take a minute to simplify what we've got so far.

Remember why we needed the `RepeaterIterator` class again? We needed it to host the `__next__` method for fetching new values from the iterator. But it doesn't really matter *where* `__next__` is defined. In the iterator protocol, all that matters is that `__iter__` returns *any* object with a `__next__` method on it.

So here's an idea: `RepeaterIterator` returns the same value over and over, and it doesn't have to keep track of any internal state. What if we added the `__next__` method directly to the `Repeater` class instead?

That way we could get rid of `RepeaterIterator` altogether and implement an iterable object with a single Python class. Let's try it out! Our new and simplified iterator example looks as follows:

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

    def __iter__(self):
        return self

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

We just went from two separate classes and 10 lines of code to to just one class and 7 lines of code. Our simplified implementation still supports the iterator protocol just fine:

```
>>> repeater = Repeater('Hello')
>>> for item in repeater:
...     print(item)

Hello
Hello
Hello
...
```

Streamlining a class-based iterator like that often makes sense. In fact, most Python iterator tutorials start out that way. But I always felt that explaining iterators with a single class from the get-go hides the underlying principles of the iterator protocol—and thus makes it more difficult to understand.

Who Wants to Iterate Forever

At this point you'll have a pretty good understanding of how iterators work in Python. But so far we've only implemented iterators that kept on iterating *forever*.

Clearly, infinite repetition isn't the main use case for iterators in Python. In fact, when you look back all the way to the beginning of this tutorial, I used the following snippet as a motivating example:

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

You'll rightfully expect this code to print the numbers `1`, `2`, and `3` and then stop. And you probably *don't* expect it to go on spamming your terminal window by printing threes forever until you mash *Ctrl+C* a few times in a wild panic…

And so, it's time to find out how to write an iterator that eventually *stops* generating new values instead of iterating forever. Because that's what Python objects typically do when we use them in a *for-in* loop.

We'll now write another iterator class that we'll call `BoundedRepeater`. It'll be similar to our previous `Repeater` example, but this time we'll want it to stop after a predefined number of repetitions.

Let's think about this for a bit. How do we do this? How does an iterator signal that it's exhausted and out of elements to iterate over? Maybe you're thinking, "Hmm, we could just return `None` from the `__next__` method."

And that's not a bad idea—but the trouble is, what are we going to do if we *want* some iterators to be able to return `None` as an acceptable value?

Let's see what other Python iterators do to solve this problem. I'm going to construct a simple container, a list with a few elements, and then I'll iterate over it until it runs out of elements to see what happens:

```
>>> my_list = [1, 2, 3]
>>> iterator = iter(my_list)

>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
```

Careful now! We've consumed all of the three available elements in the list. Watch what happens if I call `next` on the iterator again:

```
>>> next(iterator)
StopIteration
```

Aha! It raises a StopIteration exception to signal we've exhausted all of the available values in the iterator.

That's right: Iterators use exceptions to structure control flow. To signal the end of iteration, a Python iterator simply raises the built-in `StopIteration` exception.

If I keep requesting more values from the iterator it'll keep raising `StopIteration` exceptions to signal that there are no more values available to iterate over:

```
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
...
```

Python iterators normally can't be "reset"—once they're exhausted they're supposed to raise `StopIteration` every time `next()` is called on them. To iterate anew you'll need to request a fresh iterator object with the `iter()` function.

Now we know everything we need to write our `BoundedRepeater` class that stops iterating after a set number of repetitions:

```
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
```

This gives us the desired result. Iteration stops after the number of repetitions defined in the `max_repeats` parameter:

```
>>> repeater = BoundedRepeater('Hello', 3)
>>> for item in repeater:
        print(item)
Hello
Hello
Hello
```

If we rewrite this last for-in loop example to take away some of the syntactic sugar, we end up with the following expanded code snippet:

```
repeater = BoundedRepeater('Hello', 3)
iterator = iter(repeater)
while True:
    try:
        item = next(iterator)
    except StopIteration:
        break
    print(item)
```

Every time `next()` is called in this loop we check for a `StopIteration` exception and break the `while` loop if necessary.

Being able to write a three-line *for-in* loop instead of an eight lines long `while` loop is quite a nice improvement. It makes the code easier to read and more maintainable. And this is another reason why iterators in Python are such a powerful tool.

Python 2.x Compatible Iterators

All the code examples I showed here were written in Python 3. There's a small but important difference between Python 2 and 3 when it comes to implementing class-based iterators:

- In Python 3, the method that retrieves the next value from an iterator is called `__next__`.
- In Python 2, the same method is called `next` (no underscores).

This naming difference can lead to some trouble if you're trying to write class-based iterators that should work on both versions of Python. Luckily there's a simple approach you can take to work around this difference.

Here's an updated version of the `InfiniteRepeater` class that will work on both Python 2 and Python 3:

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

    def __iter__(self):
        return self

    def __next__(self):
        return self.value

    # Python 2 compatibility:
    def next(self):
        return self.__next__()
```

To make this iterator class compatible with Python 2 I've made two small changes to it:

First, I added a `next` method that simply calls the original `__next__` and forwards its return value. This essentially creates an alias for the existing `__next__` implementation

so that Python 2 finds it. That way we can support both versions of Python while still keeping all of the actual implementation details in one place.

And second, I modified the class definition to inherit from `object` in order to ensure we're creating a *new-style* class on Python 2. This has nothing to do with iterators specifically, but it's a good practice nonetheless.

Python Iterators – A Quick Summary

- Iterators provide a sequence interface to Python objects that's memory efficient and considered Pythonic. Behold the beauty of the *for-in* loop!
- To support iteration an object needs to implement the *iterator protocol* by providing the `__iter__` and `__next__` dunder methods.
- Class-based iterators are only one way to write iterable objects in Python. Also consider generators and generator expressions.