Trey Hunner

I help developers level-up their Python skills

Hire Me For Training

- RSS

Search

Navigate…

- Articles
- Talks
- Python Morsels
- Team Training
- About

How to make an iterator in Python

Jun 21st, 2018 4:00 pm | Comments

I wrote an article sometime ago on the iterator protocol that powers Python's `for` loops. One thing I left out of that article was **how to make your own iterators**.

In this article I'm going to discuss why you'd want to make your own iterators and then show you how to do so.

What is an iterator?

First let's quickly address what an iterator is. For a much more detailed explanation, consider watching my Loop Better talk or reading the article based on the talk.

An **iter*able*** is anything you're able to loop over.

An **iter*ator*** is the object that does the actual iterating.

You can get an iterator from any iterable by calling the built-in `iter` function on the iterable.

```
>>> favorite_numbers = [6, 57, 4, 7, 68, 95]
>>> iter(favorite_numbers)
<list_iterator object at 0x7fe8e5623160>
```

You can use the built-in `next` function on an iterator to get the next item from it (you'll get a `StopIteration` exception if there are no more items).

```
>>> favorite_numbers = [6, 57, 4, 7, 68, 95]
>>> my_iterator = iter(favorite_numbers)
>>> next(my_iterator)
6
>>> next(my_iterator)
57
```

There's one more rule about iterators that makes everything interesting: **iterators are also iterables** and their iterator is themselves. I explain the consequences of that more fully in that Loop Better talk I mentioned above.

Why make an iterator?

Iterators allow you to make an iterable that computes its items as it goes. Which means that you can make iterables that are **lazy**, in that they don't determine what their next item is until you ask them for it.

Using an iterator instead of a list, set, or another iterable data structure can sometimes allow us to save memory. For example, we can use `itertools.repeat` to create an iterable that provides 100 million `4`'s to us:

```
>>> from itertools import repeat
>>> lots_of_fours = repeat(4, times=100_000_000)
```

This iterator takes up 56 bytes of memory on my machine:

```
>>> import sys
>>> sys.getsizeof(lots_of_fours)
56
```

An equivalent list of 100 million `4`'s takes up many megabytes of memory:

```
>>> lots_of_fours = [4] * 100_000_000
>>> import sys
>>> sys.getsizeof(lots_of_fours)
800000064
```

While iterators can save memory, they can also save time. For example if you wanted to print out just the first line of a 10 gigabyte log file, you could do this:

```
>>> print(next(open('giant_log_file.txt')))
This is the first line in a giant file
```

File objects in Python are implemented as iterators. As you loop over a file, data is read into memory one line at a time. If we instead used the `readlines` method to store all lines in memory, we might run out of system memory.

So **iterators can save us memory**, but **iterators can sometimes save us time** also.

Additionally, **iterators have abilities that other iterables don't**. For example, the laziness of iterables can be used to make iterables that have an unknown length. In fact, you can even make infinitely long iterators.

For example, the `itertools.count` utility will give us an iterator that will provide every number from `0` upward as we loop over it:

```
>>> from itertools import count
>>> for n in count():
...     print(n)
...
0
1
2
(this goes on forever)
```

That `itertools.count` object is essentially an infinitely long iterable. And it's implemented as an iterator.

Making an iterator: the object-oriented way

So we've seen that iterators can save us memory, save us CPU time, and unlock new abilities to us.

Let's make our own iterators. We'll start be re-inventing the `itertools.count` iterator object.

Here's an iterator implemented using a class:

```
class Count:

    """Iterator that counts upward forever."""

    def __init__(self, start=0):
        self.num = start

    def __iter__(self):
        return self

    def __next__(self):
        num = self.num
        self.num += 1
        return num
```

This class has an initializer that initializes our current number to `0` (or whatever is passed in as the `start`). The things that make this class usable as an iterator are the `__iter__` and `__next__` methods.

When an object is passed to the `str` built-in function, its `__str__` method is called. When an object is passed to the `len` built-in function, its `__len__` method is called.

```
>>> numbers = [1, 2, 3]
>>> str(numbers), numbers.__str__()
('[1, 2, 3]', '[1, 2, 3]')
>>> len(numbers), numbers.__len__()
(3, 3)
```

Calling the built-in `iter` function on an object will attempt to call its `__iter__` method. Calling the built-in `next` function on an object will attempt to call its `__next__` method.

The `iter` function is supposed to return an iterator. So our `__iter__` function must return an iterator. But **our object is an iterator**, so should return ourself. Therefore our `Count` object returns `self` from its `__iter__` method because it is *its own iterator*.

The `next` function is supposed to return the next item in our iterator or raise a `StopIteration` exception when there are no more items. We're returning the current number and incrementing the number so it'll be larger during the next `__next__` call.

We can manually loop over our `Count` iterator class like this:

```
>>> c = Count()
>>> next(c)
0
>>> next(c)
1
```

We could also loop over our `Count` object like using a `for` loop, as with any other iterable:

```
>>> for n in Count():
...     print(n)
...
0
1
2
(this goes on forever)
```

This object-oriented approach to making an iterator is cool, but it's not the usual way that Python programmers make iterators. Usually when we want an iterator, we make a generator.

Generators: the easy way to make an iterator

The easiest ways to make our own iterators in Python is to create a generator.

There are two ways to make generators in Python.

Given this list of numbers:

```
>>> favorite_numbers = [6, 57, 4, 7, 68, 95]
```

We can make a generator that will lazily provide us with all the squares of these numbers like this:

```
>>> def square_all(numbers):
...     for n in numbers:
...         yield n**2
...
>>> squares = square_all(favorite_numbers)
```

Or we can make the same generator like this:

```
>>> squares = (n**2 for n in favorite_numbers)
```

The first one is called a **generator function** and the second one is called a **generator expression**.

Both of these generator objects work the same way. They both have a type of `generator` and they’re both iterators that provide squares of the numbers in our numbers list.

```
>>> type(squares)
<class 'generator'>
>>> next(squares)
36
>>> next(squares)
3249
```

We’re going to talk about both of these approaches to making a generator, but first let’s talk about terminology.

The word “generator” is used in quite a few ways in Python:

- A **generator**, also called a **generator object**, is an iterator whose type is `generator`
- A **generator function** is a special syntax that allows us to make a function which returns a **generator object** when we call it
- A **generator expression** is a comprehension-like syntax that allows you to create a **generator object** inline

With that terminology out of the way, let’s take a look at each one of these things individually. We’ll look at generator functions first.

Generator functions

Generator functions are distinguished from plain old functions by the fact that they have one or more `yield` statements.

Normally when you call a function, its code is executed:

```
>>> def gimme4_please():
...     print("Let me go get that number for you.")
...     return 4
...
>>> num = gimme4_please()
Let me go get that number for you.
>>> num
4
```

But if the function has a `yield` statement in it, it isn’t a typical function anymore. It’s now a **generator function**, meaning it will return a **generator object** when called. That generator object can be looped over to execute it until a `yield` statement is hit:

```
>>> def gimme4_later_please():
...     print("Let me go get that number for you.")
...     yield 4
...
>>> get4 = gimme4_later_please()
>>> get4
<generator object gimme4_later_please at 0x7f78b2e7e2b0>
>>> num = next(get4)
Let me go get that number for you.
>>> num
4
```

The mere presence of a `yield` statement turns a function into a generator function. If you see a function and there’s a `yield`, you’re working with a different animal. It’s a bit odd, but that’s the way generator functions work.

Okay let’s look at a real example of a generator function. We’ll make a generator function that does the same thing as our `Count` iterator class we made earlier.

```
def count(start=0):
    num = start
    while True:
        yield num
        num += 1
```

Just like our `Count` iterator class, we can manually loop over the generator we get back from calling `count`:

```
>>> c = count()
>>> next(c)
0
>>> next(c)
1
```

And we can loop over this generator object using a `for` loop, just like before:

```
>>> for n in count():
...     print(n)
...
0
1
2
(this goes on forever)
```

But this function is considerably shorter than our `Count` class we created before.

Generator expressions

Generator expressions are a list comprehension-like syntax that allow us to make a generator object.

Let’s say we have a list comprehension that filters empty lines from a file and strips newlines from the end:

```
lines = [
    line.rstrip('\n')
    for line in poem_file
    if line != '\n'
]
```

We could create a generator instead of a list, by turning the square brackets of that comprehension into parenthesis:

```
lines = (
    line.rstrip('\n')
    for line in poem_file
    if line != '\n'
)
```

Just as our list comprehension gave us a list back, our **generator expression** gives us a **generator object** back:

```
>>> type(lines)
<class 'generator'>
>>> next(lines)
' This little bag I hope will prove'
>>> next(lines)
'To be not vainly made--'
```

Generator expressions use a shorter inline syntax compared to generator functions. They’re not as powerful though.

If you can write your generator function in this form:

```
def get_a_generator(some_iterable):
    for item in some_iterable:
        if some_condition(item):
            yield item
```

Then you can replace it with a generator expression:

```
def get_a_generator(some_iterable):
    return (
        item
        for item in some_iterable
        if some_condition(item)
    )
```

If you can’t write your generator function in that form, then you can’t create a generator expression to replace it.

Generator expressions vs generator functions

You can think of generator expressions as the list comprehensions of the generator world.

If you’re not familiar with list comprehensions, I recommend reading my article on list comprehensions in Python. I note in that article that you can copy-paste your way from a `for` loop to a list comprehension.

You can also copy-paste your way from a generator function to a function that returns a generator expression:

```
def cubes_of_odds(numbers):
    for n in numbers:
        if n % 2 == 1:
            yield n**3
```

Generator expressions are to generator functions as list comprehensions are to a simple `for` loop with an append and a condition.

Generator expressions are so similar to comprehensions, that you might even be tempted to say **generator comprehension** instead of generator expression. That’s not technically the correct name, but if you say it everyone will know what you’re talking about. Ned Batchelder actually proposed that we should all start calling generator expressions generator comprehensions and I tend to agree that this would be a clearer name.

So what’s the best way to make an iterator?

To make an iterator you could create an iterator class, a generator function, or a generator expression. Which way is the best way though?

Generator expressions are **very succinct**, but they’re **not nearly as flexible** as generator functions. Generator functions are flexible, but if you need to **attach extra methods or attributes** to your iterator object, you’ll probably need to switch to using an iterator class.

I’d recommend reaching for generator expressions the same way you reach for list comprehensions. If you’re doing a simple **mapping or filtering operation**, a **generator expression** is a great solution. If you’re doing something **a bit more sophisticated**, you’ll likely need a **generator function**.

I’d recommend using generator functions the same way you’d use `for` loops that append to a list. Everywhere you’d see an `append` method, you’d often see a `yield` statement instead.

And I’d say that you should **almost never create an iterator class**. If you find you need an iterator class, try to write a generator function that does what you need and see how it compares to your iterator class.

Generators can help when making iterables too

You’ll see iterator classes in the wild, but there’s rarely a good opportunity to write your own.

While it’s rare to create your own iterator class, it’s not as unusual to make your own iterable class. And iterable classes require a `__iter__` method which returns an iterator. Since generators are the easy way to make an iterator, we can use a generator function or a generator expression to create our `__iter__` methods.

For example here’s an iterable that provides x-y coordinates:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __iter__(self):
        yield self.x
        yield self.y
```

Note that our `Point` class here creates an **iterable** when called (not an iterator). That means our `__iter__` method must return an iterator. The easiest way to create an iterator is by making a generator function, so that's just what we did.

We stuck `yield` in our `__iter__` to make it into a generator function and now our `Point` class can be looped over, just like any other iterable.

```
>>> p = Point(1, 2)
>>> x, y = p
>>> print(x, y)
1 2
>>> list(p)
[1, 2]
```

Generator functions are a natural fit for creating `__iter__` methods on your iterable classes.

Generators are *the* way to make iterators

Dictionaries are the typical way to make a mapping in Python. Functions are the typical way to make a callable object in Python. Likewise, **generators are the typical way to make an iterator in Python**.

So when you're thinking "it sure would be nice to implement an iterable that lazily computes things as it's looped over," think of iterators.

And when you're considering **how to create your own iterator**, think of **generator functions** and **generator expressions**.

Practice making an iterator right now

You won't learn new Python skills by reading, you'll learn them by writing code.

If you'd like to practice making an iterator right now, sign up for Python Morsels using the form below and I'll immediately give you an exercise to practice making an iterator.

Your email

Get my iterator practice exercise

You can find the Privacy Policy for Python Morsels here.

Posted by Trey Hunner Jun 21st, 2018 4:00 pm favorite, python

Tweet

« How to have a great first PyCon Overusing lambda expressions in Python »

Comments

9 Comments Trey's Blog 1 Login

Recommend 6 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH OR SIGN UP WITH DISQUS ?

Name

Evgeniy • a year ago

Thank you, this article is the most clear explanation Python generators and iterators ever read. I almost understood this staff))

2 • Reply • Share ›

Trey Hunner Mod → Evgeniy • a year ago

Thanks for the kind words Evgeniy! I'm glad you found my explanations clear. 😊

I know I didn't cover everything, but it's hard to know what important things I left out. If you have any questions on them, please feel free to ask.

• Reply • Share ›

Tina Bu • 7 months ago

Came here after this week's parse_ranges problem. I didn't know that changing [] of list comprehension to () you get an iterator instead! However returning

```
return (num for num in results) # results is a list
```

seems to generate correct results locally but is not passing the auto grader. Looking forward to the solution I guess!

• Reply • Share ›

Ajinkya • 7 months ago

Really nice article! Thanks for making the iterator and generator concepts very clear.

• Reply • Share ›

Edmund • 9 months ago

Another fantastically clear read! Many thanks.

• Reply • Share ›

Andy Gimblett • a year ago • edited

When you say:

> There's one more rule about iterators that makes everything interesting: iterators are also iterables and their iterator is themselves. I explain the consequences of that more fully in that Loop Better talk I mentioned above.

... that's not true. That's a common convention, but it's absolutely not a rule.

Here's a trivial iterator which is not an iterable.

```
class IteratorNotIterable(object):
    def next(self):
        return 1

x = IteratorNotIterable()

# Next line works because it's an iterator.
ok = next(x)

# Next line fails because it's not an iterable.
fails = iter(x)
```

An iterable defines an `__iter__` method that returns an iterator; an iterator defines a `next` method: that's it.

• Reply • Share ›

Trey Hunner Mod → Andy Gimblett • a year ago

It's not enforced by CPython (lazy duck typing in a way) but it is required per the documentation: https://docs.python.org/3/l...

• Reply • Share ›

Andy Gimblett → Trey Hunner • a year ago

Huh. TIL. Thanks!

• Reply • Share ›

Trey Hunner Mod → Andy Gimblett • a year ago

You're welcome! I think it's easy to overlook that definition because not all programming languages define iterators that way.

• Reply • Share ›

ALSO ON **TREY'S BLOG**

Keyword (Named) Arguments in Python: How to Use Them

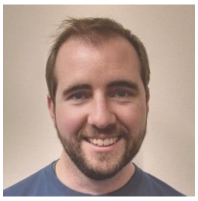
14 comments • 2 years ago

AvatarBoutros Gladius — Excellent exposition, ta.

The Iterator Protocol: How for Loops Work in Python

19 comments • 3 years ago

AvatarChristina — Thanks for the article, it explains then iterators accurately. As I am new to Python, I would like to ask: does



Hi! My name is Trey Hunner.

I help Python teams **write better Python code** through Python team training.

I also help individuals **level-up their Python skills** with weekly Python skill-building.

Python Team Training

Write Pythonic code

The best way to improve your skills is to **write more code**, but it's time consuming to figure out what code to write. I've made a Python skill-building service to help solve this problem.

Each week you'll get an exercise that'll help you dive deeper into Python and carefully **reflect on your own coding style**. The first 4 exercises are free.

Sign up below for **four free exercises**!

Your email

Sign up

You can find the Privacy Policy here.

Favorite Posts

- Python List Comprehensions
- How to Loop With Indexes in Python
- Check Whether All Items Match a Condition in Python
- Keyword (Named) Arguments in Python: How to Use Them
- Tuple unpacking improves Python code readability
- The Idiomatic Way to Merge Dictionaries in Python
- The Iterator Protocol: How for Loops Work in Python
- Craft Your Python Like Poetry
- Python: range is not an iterator!
- Counting Things in Python: A History

Follow @treyhunner