Sejal Jaiswal
January 23rd, 2018

PYTHON

Python Iterator Tutorial

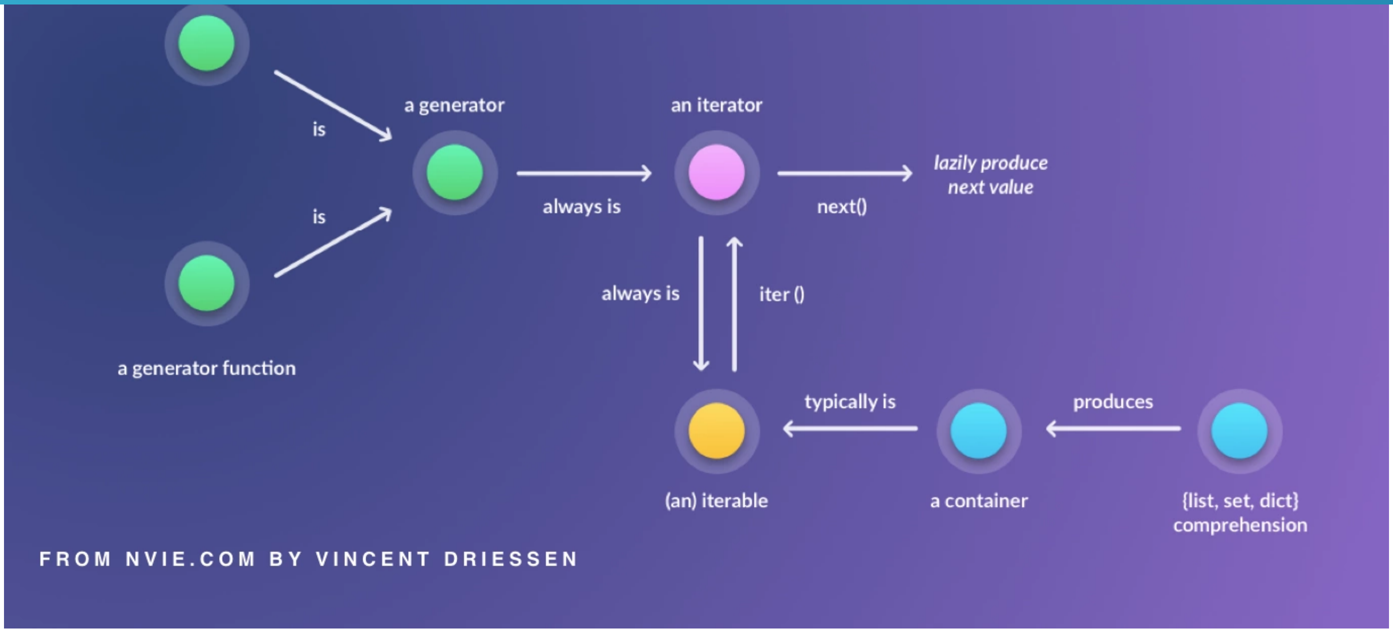
Learn all about the Python iterator, how they differ from iterables and generators, and how to build one yourself with __iter__, __next__ and itertools.

Iterators are the omnipresent spirits of Python. They are everywhere and you must have come across them in some program or another. Iterators are objects that allow you to traverse through all the elements of a collection, regardless of its specific implementation.

That means that, if you have ever used loops to iterate or run through the values in a container, you have used an iterator.

In this post, you will learn more about Python iterators. More specifically, you will

- First you'll see Iterators in detail to really understand what they are about and when you should use them.
- Then, you will see what Iterables are, since there is an important difference between the two!
- Next, you'll learn about Containers and how they use the concept of iterators.
- You will then see the Itertools Module in action.



Source: Iterables vs. Iterators vs. Generators by Vincent Driessen

Be sure to check out DataCamp's two-part Python Data Science ToolBox course. The second part will work you through iterators, loops and list comprehension. It is followed by a case study in which you will apply all of the techniques you learned in the course: part 1 and part 2 combined.

Now, let's dive into iterators.....

Iterators

An iterator is an object that implements the **iterator protocol** (don't panic!). An iterator protocol is nothing but a specific class in Python which further has the `__next()__` method. Which means every time you ask for the next value, an iterator knows how to compute it. It keeps information about the current state of the iterable it is working on. The iterator calls the next value when you call `next()` on it. An object that uses the `__next__()` method is ultimately an iterator.

Iterators help to produce cleaner looking code because they allows us to work with infinite

Want to leave a comment?

iterators. You will see what `itertools` are later on in this tutorial.

Iterables

According to Vincent Driessen of nvie.com, "an iterable is any object, not necessarily a data structure that can return an iterator". Its main purpose is to return all of its elements. Iterables can represent finite as well as infinite source of data. An iterable will directly or indirectly define two methods: the `__iter__()` method, which must return the iterator object and the `__next()__` method with the help of the iterator it calls.

Note: Often the iterable classes will implement both `__iter__()` and `__next__()` in the same class, and have `__iter__()` return self, which makes the `_iterable_` class both an iterable and its own iterator. It's perfectly fine to return a different object as the iterator, though.

There is an major dissimilarity between what an iterable is and what an iterator is. Here is an example:

```
a_set = {1, 2, 3}
b_iterator = iter(a_set)
next(b_iterator)
```

```
type(a_set)
```

```
type(b_iterator)
```

In the example, `a_set` is an iterable (a set) whereas `b_iterator` is an iterator. They are both different data types in Python.

Wondering how an iterator works internally to produce the next sequence when asked? Let's

Want to leave a comment?

DataCamp

```
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

n_list = Series(1,10)
print(list(n_list))
```

`__iter__` returns the iterator object itself and the `__next__` method returns the next value from the iterator. If there is no more items to return then it raises a `StopIteration` exception.

It is perfectly fine if you cannot write the code for an iterator yourself at this moment, but it is important that you grasp the basic concept behind it. You will see generators later on in the tutorial, which is a much easier way of implementing iterators.

Containers

Containers are the objects that hold data values. They support membership tests, which means you can check if a value exists in the container. Containers are iterables - lists, sets, dictionary, tuple and strings are all containers. But there are other iterables as well like open files and open sockets. You can perform membership tests on the containers:

Want to leave a comment?

```
if 'apple' in 'pineapple':
    print('String') #string contains all its substrings
```

```
List
Tuple
String
```

Itertools Module

`Itertools` is an built-in Python module that contains functions to create iterators for efficient looping. In short, it provides a lot of interesting tools to work with iterators! Some keep providing values for an infinite range, hence they should only be accessed by functions or loops that actually stop calling for more values eventually.

Let's check out some cool things that you can do with the `count` function from the `itertools` module:

```
from itertools import count
sequence = count(start=0, step=1)
while(next(sequence) <= 10):
    print(next(sequence))
```

```
1
3
5
7
9
11
```

```
from itertools import cycle
```

Want to leave a comment?

```
Q. What do we have for dessert? A: Icecream
Q. What do we have for dessert? A: Cake
Q. What do we have for dessert? A: Icecream
Q. What do we have for dessert? A: Cake
```

You can learn more in depth about the `itertools` here.

Generators

The generator is the elegant brother of iterator that allows you to write iterators like the one you saw earlier, but in a much easier syntax where you do not have to write classes with `__iter__()` and `__next__()` methods.

Remember the example that you saw earlier with the iterator? Let's try to rewrite the code but using the concept of generators:

```
def series_generator(low, high):
    while low <= high:
       yield low
       low += 1

n_list = []
for num in series_generator(1,10):
    n_list.append(num)

print(n_list)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Want to leave a comment?

while loop and comes to the `yield` statement again.

`yield` basically replaces the return statement of a function but rather provides a result to its caller without destroying local variables. Thus, in the next iteration, it can work on this local variable value again. So unlike a normal function that you have seen before, where on each call it starts with new set of variables - a generator will resume the execution where it was left off.

Tip: lazy factory is a concept behind the generator and the iterator. Which means they are idle until you ask it for a value. Only when asked is when they get to work and produce a single value, after which it turns idle again. This is a good approach to work with lots of data. If you do not require all the data at once and hence no need to load all the data in the memory, you can use a generator or an iterator which will pass you each piece of data at a time.

Types of Generators

Generators can be of two different types in Python: generator functions and generator expressions.

A generator function is a function where the keyword `yield` appears in the body. You have already seen an example of this with the `series_generator` function. Which means the appearance of the keyword `yield` is enough to make the function a generator function.

The generator expressions are the generator equivalent of a list comprehension. They can be specially useful for a limited use case. Just like a list comprehension returns a list, a generator expressions will return a generator.

Let's see what this means:

```
squares = (x * x for x in range(1,10))
print(type(squares))
```

Want to leave a comment?

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

The power of Generators is extreme. They are more memory and CPU efficient and allow you to write code with fewer intermediate variables and data structures. They usually require fewer lines of code and their usage makes the code easier to read and understand. That's why it's important that you try to use generators in your code as much as possible (Iterables vs. Iterators vs. Generators).

Where can you insert generators in your code? **Tip**: find places in your code where you do the following:

```
def some_function():
    result = []
    for ... in ...:
        result.append(x)
    return result
```

And replace it with:

```
def iterate_over():
    for ... in ...:
        yield x
```

Well done, Pythonista!

Iterators are a powerful and useful tool in Python. However, not everyone is very familiar with the nitty-gritty details of it. Congrats on making it to the end of this tutorial!

Head over to DataCamp's Intermediate Python for Data Science course. This tutorial works with `matplotlib`'s functions that you can use to visualize real data. You will also learn about new data structures such as the dictionary and the Pandas DataFrame. You have already seen

Want to leave a comment?

DataCamp

- Iterables vs. Iterators vs. Generators by Vincent Driessen
- Iterators, generators and decorators

▲ 8 3

COMMENTS

Johan Haggle von Neubrücke
29/11/2018 01:01 PM

From the article above:

"An iterator is an object that implements the **iterator protocol** (don't panic!)."

"Python has several built-in objects, which implement the iterator protocol and you must have seen some of these before: lists, tuples, strings, dictionaries and even files."

This implies that a list is an iterator, but I learn from otherwhere that it is not. The article is somewhat vague and unclear and should be updated.

▲ 5 REPLY

Somnath Roy
27/07/2019 12:09 AM

I am glad i read your comment before reading the article.

▲ 1 REPLY

vipul sonawane
22/01/2020 11:31 PM

Want to leave a comment?

DataCamp

Subscribe to RSS

About Terms Privacy

Want to leave a comment?