Part 3: Iterators and Iterables

Bookmark this page

Part 3: Iterators and Iterables

Iterators and Iterables are key language features of Python 3, that help to make code easy to write and maintain.

First, we are going to look at some background from Python 2, before moving on to cover the main topic.

Background

Python used to be all about sequences — a good chunk of anything you did was stored in a sequence or involved manipulating one.

- lists
- tuples
- strings
- dict.keys()
- dict.values()
- dict.items()
- zip()

In **Python2** those are all sequences. It turns out, however, that the most common operation for sequences is to iterate through them:

```
for item in a_sequence:
    do_something_with_item
```

So fairly early in Python2, Python introduced the idea of the "iterable". An iterable is something you can, well, iterate over in a for loop, but often does not keep the whole sequence in memory at once. After all, why make a copy of something just to look at all its items?
For example, in python2: dict.keys() returns a list of all the keys in the dict. But why make a full copy of all the keys, when all you want to do is:

```
for key in dict.keys():
    do_something_with(key)
```

Even worse `dict.items()` created a full list of `(key,value)` tuples — a complete copy of all the data in the dict. Yet worse `enumerate(dict.items())` created a whole list of
`(index, (key, value))` tuples — lots of copies of everything.
Python2 then introduced "iterable" versions of a number of functions and methods:
itertools.izip
dict.iteritems()
dict.iterkeys()
dict.itervalues()
So you could now iterate through that stuff without copying anything.

Python 3

Python3 embraces iterables — now everything that can be an iterator is already an iterator — no unnecessary copies.
An iterator is an iterable that has been made more efficient by removing as much from memory as possible. Therefore, if you need a list, you have to make the list explicitly, as in:

```
list(dict.keys())
```

Also, there is an entire module:`itertools` that provides nifty ways to iterate through stuff. So, while we used to think in terms of sequences, we can now think in terms of iterables.

Let's see an example of how Iteration makes Python code so readable:

```
for x in just_about_anything:
    do_stuff(x)
```

An iterable is anything that can be looped over sequentially, so it does not have to be a "sequence": list, tuple, etc. For

example, a string is iterable. So is a set.
An iterator is an iterable that remembers state. All sequences are iterable, but not all sequences are iterators. To make a sequence an iterator, you can call it with iter:

```
my_iter = iter(my_sequence)
```

Iterables

To make an object iterable, you simply have to implement the __getitem__ method.

```
class T:
    def __getitem__(self, position):
        if position > 5:
         return position
```

iter()
How do you get the iterator object from an “iterable”? The iter() function will make any iterable an iterator. It first looks for the __iter__() method, and if none is found, uses get_item to create the iterator. The iter() function:

```
In []: iter([2,3,4])
Out[]: <listiterator at 0x101e01350>
In []: iter("a string")
Out[]: <iterator at 0x101e01090>
In []: iter( ('a', 'tuple') )
Out[]: <tupleiterator at 0x101e01710>
```

List as an Iterator

```
In []: a_list = [1,2,3]
In []: list_iter = iter(a_list)
In []: next(list_iter)
Out[]: 1
In []: next(list_iter)
Out[]: 2
In []: next(list_iter)
Out[]: 3
In []: next(list_iter)
--------------------------------------------------
StopIteration     Traceback (most recent call last)
<ipython-input-15-1a7db9b70878> in <module>()
----> 1 next(list_iter)
StopIteration:
```

Use iterators when you can

Consider the example from the trigrams problem: (http://codekata.com/kata/kata14-tom-swift-under-the-milkwood/)
You have a list of words and you want to go through it, three at a time, and match up pairs with the following word. The *non-pythonic* way to do that is to loop through the indices:

```

for i in range(len(words)-2):
    triple = words[i:i+3]
```

It works, and is fairly efficient, but what about:

```
    for triple in zip(words[:-2], words[1:-1], words[2:-2]):

 zip() returns an iterable --- it does not build up the whole list, so
 this is quite efficient.  However, we are still slicing: ([1:]), which
 produces a copy --- so we are creating three copies of the list ---
 not so good if memory is tight.  Note that they are shallow copies, so
 this is not terribly bad.  Nevertheless, we can do better.

The ``itertools`` module has a ``islice()`` (iterable slice)
 function.  It returns an iterator over a slice of a sequence --- so no
 more copies:
   from itertools import islice
   triplets = zip(words, islice(words, 1, None), islice(words, 2,
None))
   for triplet in triplets:
       print(triplet)
   ('this', 'that', 'the')
   ('that', 'the', 'other')
   ('the', 'other', 'and')
   ('other', 'and', 'one')
   ('and', 'one', 'more')
```

The Iterator Protocol

The main thing that differentiates an iterator from an iterable (sequence) is that an iterator saves state. An iterable must have the following methods:

an_iterator.__iter__()

Usually returns the iterator object itself.

an_iterator.__next__()

returns the next item from the container. If there are no further items it raises the `StopIteration` exception.

Making an Iterator

A simple version of `range()`

```
class IterateMe_1:
    def __init__(self, stop=5):
        self.current = 0
        self.stop = stop
    def __iter__(self):
        return self
    def __next__(self):
        if self.current < self.stop:
            self.current += 1
            return self.current
        else:
            raise StopIteration
```

What does *for* do?

Now that we know the iterator protocol, we can write something like a for loop:

```
def my_for(an_iterable, func):
    """
    Emulation of a for loop.
    func() will be called with each item in an_iterable
    """
    # equiv of "for i in l:"
    iterator = iter(an_iterable)
    while True:
        try:
            i = next(iterator)
        except StopIteration:
            break
        func(i)
```

Summary