An **iterable** object is an object that implements `__iter__`, which is expected to return an **iterator** object.

An **iterator** object implements `__next__`, which is expected to return the next element of the iterable object that returned it, and to raise a `StopIteration` exception when no more elements are available.

In the simplest case, the iterable will implement `__next__` itself and return `self` in `__iter__`. However, this has its limitations and may produce unexpected results in concurrent environments (e.g. with the multiprocessing API).

You can use iterables in for loops, to construct lists with list comprehensions, or as input arguments for the `list` function.

Example Iterator

Here is an iterator that returns a random number of 1's:

Toggle line numbers

```
1 import random
2
3 class RandomIterable:
```

```
    def __iter__(self):
        return self
    def __next__(self):
        if random.choice(["go", "go", "stop"]) == "stop":
            raise StopIteration  # signals "the end"
        return 1
```

Q: Why is `__iter__` there, if it just returns self?

A: This is a very simple case. More complex iterables may very well return separate iterator objects.

Q: When would I need an extra iterator?

A: Iterators will typically need to maintain some kind of position state information (e.g., the index of the last element returned). If the iterable maintained that state itself, it would become inherently non-reentrant (meaning you could use it only one loop at a time).

Toggle line numbers

```
for eggs in RandomIterable():
    print(eggs)
```

You can also use it in list construction:

Toggle line numbers

```
>>> list(RandomIterable())
[1]
>>> list(RandomIterable())
[]
>>> list(RandomIterable())
[1, 1, 1, 1, 1]
>>> list(RandomIterable())
[1]
```

...both of these uses require `__iter__`.

An object isn't iterable unless it provides `__iter__`. And for an object to be a valid iterator, it must provide `__next__`.

Manual usage

Although you won't need this in most cases, you can manually get the iterator from an iterable object by using the `iter()` function. Similary, you can manually call `__next___` using the `next()` function.

Links

- PEP-234: Iterators
- Itertools: Functions creating iterators for efficient looping
- Functional programming and iterators

» Python iterator basics (how they work + examples)

See also: Generators

Iterator (last edited 2021-12-11 12:50:13 by ChrisM)