Part 5: Recap

Bookmark this page

Part 5: Recap

Iterator

An iterator is a stateful helper object that will produce the next value when you call next() on it. Any object that has a __next__() method is therefore an iterator. How it produces a value is irrelevant. Think of an iterator as a value factory. Each time you ask it for “the next” value, it knows how to compute it because it holds internal state.

Iterables

Iterable objects are objects that conform to the “Iteration Protocol” and can hence be used in a loop. They always return an iterator.

This protocol consists in two methods: * the “__iter__” method that returns the object we would to iterate over and * the “__next__” method that is called automatically on each iteration and that returns the value for the current iteration.

Often, for pragmatic reasons, iterable classes will implement both __iter__() and __next__() in the same class, and have __iter__() return self , which makes the class both an iterable and its own iterator. It is perfectly fine to return a different object as the iterator though.

```
# here range() is an iterable object...
# at each iteration i is given a different value
# values returned one value at a time

for i in range(50):
    print(i)
```

You can create your own iterable by implementing the iteration protocol.

```
class fibonacci:
    def __init__(self, max=1000000):
        self.a, self.b = 0, 1
        self.max = max
    def __iter__(self):
        # Return the iterable object (self)
        return self
    def __next__(self):
        # To stop the iteration we just need to raise
        # a StopIteration exception
        if self.a > self.max:
            raise StopIteration
        # save the value that has to be returned
        value_to_be_returned = self.a
        # calculate the next values of the sequence
        self.a, self.b = self.b, self.a + self.b
        return value_to_be_returned

if __name__ == '__main__':
    MY_FIBONACCI_NUMBERS = fibonacci()
    for fibonacci_number in MY_FIBONACCI_NUMBERS:
        print(fibonacci_number)
```

and now let’s prove an iterator is stateful...

```
"""
Simple iterator examples
"""


class IterateMe_1:
    """
```

```
    returns a sequence of numbers
    ( like range() )
    """

    def __init__(self, start, increment, stop):
        self.start = start
        self.increment = increment
        self.stop = stop
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        self.current += self.increment
        if self.current < self.stop:
            return self.current
        else:
            raise StopIteration

if __name__ == "__main__":

    iter = IterateMe_1(5,2,17)
    for i in iter:
        if i >10: break
        print(i)
    # it's stateful
    for i in iter:
        print(i)

    # reinitialize "loses state"
    iter = IterateMe_1(5,2,17)
    for i in iter:
        print(i)
```

Generators

Generators in Python are just another way of creating iterable objects. They are normally used when you need to create an iterable object quickly, without the need of creating a class and adopting the iteration protocol. They are “just a function” (or a comprehension). They are used once; to use them subsequent times you have to call the generator again. An iterator is usually more memory-efficient than a generator, though. And, somewhat related, generators can be faster. BUT MEASURE!

```
def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b
if __name__ == '__main__':
    # Create generator of fibonacci numbers
    fibonacci_generator = fibonacci(1000000)
    # print out all the sequence
    for fibonacci_number in fibonacci_generator:
        print(fibonacci_number)
```

This has one yield statement, but a generator can have several. For each yield the state of the generator is saved. All generators are iterators (but not vice versa); that is, Generator in Python is a subclass of Iterator. Try to prove this (hint issubclass() function).

Itertools

Purpose: Itertools provides a set of memory efficient but fast tools, that can be used to provide comprehensive looping features in pure Python. Examples include counting, repetition and grouping.

‹ Previous Next ›

Example

.

chain:

```
import itertools
for i in itertools.chain('ABC', 'DEF'):
    print(i)
```

count:

```
from itertools import *
for i in islice(count(), 5, 10):
    print(i)
```

permutations:

```
from itertools import *
for i in permutations("ABC"):
    print(i)
```