Difference between Python Generators vs Iterators

What is the difference between iterators and generators? Some examples for when you would use each case would be helpful.

python

asked May 5 '10 at 21:14

newToProgramming
716 2 7 8

add a comment

2 Answers

`iterator` is a more general concept: any object whose class has a `next` method (`__next__` in Python 3) and an `__iter__` method that does `return self` .

Every generator is an iterator, but not vice versa. A generator is built by calling a function that has one or more `yield` expressions (`yield` statements, in Python 2.5 and earlier), and is an object that meets the previous paragraph's definition of an `iterator` .

You may want to use a custom iterator, rather than a generator, when you need a class with somewhat complex state-maintaining behavior, or want to expose other methods besides `next` (and `__iter__` and `__init__`). Most often, a generator (sometimes, for sufficiently simple needs, a generator *expression*) is sufficient, and it's simpler to code because state maintenance (within reasonable limits) is basically "done for you" by the frame getting suspended and resumed.

For example, a generator such as:

```
def squares(start, stop):
    for i in xrange(start, stop):
        yield i * i

generator = squares(a, b)
```

or the equivalent generator expression (genexp)

```
generator = (i*i for i in xrange(a, b))
```

would take more code to build as a custom iterator:

```
class Squares(object):
    def __init__(self, start, stop):
       self.start = start
       self.stop = stop
    def __iter__(self): return self
    def next(self):
       if self.start >= self.stop:
           raise StopIteration
       current = self.start * self.start
       self.start += 1
       return current

iterator = Squares(a, b)
```

But, of course, with class `Squares` you could easily offer extra methods, i.e.

```
    def current(self):
       return self.start
```

if you have any actual need for such extra functionality in your application.

edited Oct 16 at 19:21
Søren Løvborg
3,619 18 26

answered May 5 '10 at 21:19
Alex Martelli
358k 56 680 993

Great explanation. However, I don't think that generators are iterators. I believe that they are named generators because they generate iterators. – Tyler Crompton Dec 26 '12 at 17:20

12 @TylerCrompton generators are in fact iterators. They possess `__next__` methods that return the next item and `__iter__` methods that return the generator itself. – tjd.rodgers Dec 27 '12 at 23:42

@tjd.rodgers, ah, jumped the gun and didn't do a simple `dir` to check. – Tyler Crompton Dec 28 '12 at 1:15

more python3 notes, available in Python 2.6 or greater, `next(obj)` replaces `obj.next()` -- see also PEP 3114 and theres-no-next-function-in-a-yield-generator-in-python-3 – here May 4 at 4:06

Awesome answer--clear yet concise, explains the 'why' yet also illustrates the 'how'. – Jon Coombs Dec 11 at 21:27

add a comment

Iterators:

Iterator are objects which uses `next()` method to get next value of sequence.

Generators:

A generator is a function that produces or yields a sequence of values using `yield` method.

Every `next()` method call on generator object(for ex: `f` as in below example) returned by generator function(for ex: `foo()` function in below example), generates next value in sequence.

When a generator function is called, it returns an generator object without even beginning execution of the function. When `next()` method is called for the first time, the function starts executing until it reaches yield statement which returns the yielded value. The yield keeps track of i.e. remembers last execution. And second `next()` call continues from previous value.

The following example demonstrates the interplay between yield and call to next method on generator object.

```
>>> def foo():
...     print "begin"
...     for i in range(3):
...         print "before yield", i
...         yield i
...         print "after yield", i
...     print "end"
...
>>> f = foo()
>>> f.next()
begin
before yield 0
0
>>> f.next()
after yield 0
before yield 1
1
>>> f.next()
after yield 1
before yield 2
2
>>> f.next()
after yield 2
end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

answered May 19 at 19:05
daa
2,322 15 35

add a comment

Not the answer you're looking for? Browse other questions tagged python or ask your own question.