CHAPTER 14

Iterables, Iterators, and Generators

When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough—often that I'm generating by hand the expansions of some macro that I need to write.[1]

— Paul Graham
Lisp hacker and venture capitalist

Iteration is fundamental to data processing. And when scanning datasets that don't fit in memory, we need a way to fetch the items *lazily*, that is, one at a time and on demand. This is what the Iterator pattern is about. This chapter shows how the Iterator pattern is built into the Python language so you never need to implement it by hand.

Python does not have macros like Lisp (Paul Graham's favorite language), so abstracting away the Iterator pattern required changing the language: the `yield` keyword was added in Python 2.2 (2001).[2] The `yield` keyword allows the construction of generators, which work as iterators.

Every generator is an iterator: generators fully implement the iterator interface. But an iterator—as defined in the GoF book—retrieves items from a collection, while a generator can produce items "out of thin air." That's why the Fibonacci sequence generator is a common example: an infinite series of numbers cannot be stored in a collection. However, be aware that the Python community treats *iterator* and *generator* as synonyms most of the time.

1. From "Revenge of the Nerds", a blog post.
2. Python 2.2 users could use `yield` with the directive `from __future__ import generators`; `yield` became available by default in Python 2.3.

WOW! eBook
www.wowebook.org

Python 3 uses generators in many places. Even the `range()` built-in now returns a generator-like object instead of full-blown lists like before. If you must build a `list` from `range`, you have to be explicit (e.g., `list(range(100))`).

Every collection in Python is *iterable*, and iterators are used internally to support:

- `for` loops
- Collection types construction and extension
- Looping over text files line by line
- List, dict, and set comprehensions
- Tuple unpacking
- Unpacking actual parameters with `*` in function calls

This chapter covers the following topics:

- How the `iter(…)` built-in function is used internally to handle iterable objects
- How to implement the classic Iterator pattern in Python
- How a generator function works in detail, with line-by-line descriptions
- How the classic Iterator can be replaced by a generator function or generator expression
- Leveraging the general-purpose generator functions in the standard library
- Using the new `yield from` statement to combine generators
- A case study: using generator functions in a database conversion utility designed to work with large datasets
- Why generators and coroutines look alike but are actually very different and should not be mixed

We'll get started studying how the `iter(…)` function makes sequences iterable.

Sentence Take #1: A Sequence of Words

We'll start our exploration of iterables by implementing a `Sentence` class: you give its constructor a string with some text, and then you can iterate word by word. The first version will implement the sequence protocol, and it's iterable because all sequences are iterable, as we've seen before, but now we'll see exactly why.

Example 14-1 shows a `Sentence` class that extracts words from a text by index.

WOW! eBook
www.wowebook.org

Example 14-1. sentence.py: A Sentence as a sequence of words

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)  ❶

    def __getitem__(self, index):
        return self.words[index]  ❷

    def __len__(self):  ❸
        return len(self.words)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)  ❹
```

❶ `re.findall` returns a list with all nonoverlapping matches of the regular expression, as a list of strings.

❷ `self.words` holds the result of `.findall`, so we simply return the word at the given index.

❸ To complete the sequence protocol, we implement `__len__`—but it is not needed to make an iterable object.

❹ `reprlib.repr` is a utility function to generate abbreviated string representations of data structures that can be very large.[3]

By default, `reprlib.repr` limits the generated string to 30 characters. See the console session in Example 14-2 to see how `Sentence` is used.

Example 14-2. Testing iteration on a Sentence instance

```
>>> s = Sentence('"The time has come," the Walrus said,')  # ❶
>>> s
Sentence('"The time ha... Walrus said,')  # ❷
>>> for word in s:  # ❸
...     print(word)
The
time
has
come
```

3. We first used `reprlib` in "Vector Take #1: Vector2d Compatible" on page 276.

WOW! eBook
www.wowebook.org

```
the
Walrus
said
>>> list(s)  # ❹
['The', 'time', 'has', 'come', 'the', 'Walrus', 'said']
```

❶ A sentence is created from a string.

❷ Note the output of `__repr__` using `...` generated by `reprlib.repr`.

❸ `Sentence` instances are iterable; we'll see why in a moment.

❹ Being iterable, `Sentence` objects can be used as input to build lists and other iterable types.

In the following pages, we'll develop other `Sentence` classes that pass the tests in Example 14-2. However, the implementation in Example 14-1 is different from all the others because it's also a sequence, so you can get words by index:

```
    >>> s[0]
    'The'
    >>> s[5]
    'Walrus'
    >>> s[-1]
    'said'
```

Every Python programmer knows that sequences are iterable. Now we'll see precisely why.

Why Sequences Are Iterable: The iter Function

Whenever the interpreter needs to iterate over an object `x`, it automatically calls `iter(x)`.

The `iter` built-in function:

1. Checks whether the object implements `__iter__`, and calls that to obtain an iterator.
2. If `__iter__` is not implemented, but `__getitem__` is implemented, Python creates an iterator that attempts to fetch items in order, starting from index 0 (zero).
3. If that fails, Python raises `TypeError`, usually saying "*C* object is not iterable," where `C` is the class of the target object.

That is why any Python sequence is iterable: they all implement `__getitem__`. In fact, the standard sequences also implement `__iter__`, and yours should too, because the special handling of `__getitem__` exists for backward compatibility reasons and may be gone in the future (although it is not deprecated as I write this).

As mentioned in "Python Digs Sequences" on page 310, this is an extreme form of duck typing: an object is considered iterable not only when it implements the special method

WOW! eBook
www.wowebook.org

`__iter__`, but also when it implements `__getitem__`, as long as `__getitem__` accepts `int` keys starting from `0`.

In the goose-typing approach, the definition for an iterable is simpler but not as flexible: an object is considered iterable if it implements the `__iter__` method. No subclassing or registration is required, because `abc.Iterable` implements the `__subclasshook__`, as seen in "Geese Can Behave as Ducks" on page 338. Here is a demonstration:

```
>>> class Foo:
...     def __iter__(self):
...         pass
...
>>> from collections import abc
>>> issubclass(Foo, abc.Iterable)
True
>>> f = Foo()
>>> isinstance(f, abc.Iterable)
True
```

However, note that our initial `Sentence` class does not pass the `issubclass(Sentence, abc.Iterable)` test, even though it is iterable in practice.

As of Python 3.4, the most accurate way to check whether an object `x` is iterable is to call `iter(x)` and handle a `TypeError` exception if it isn't. This is more accurate than using `isinstance(x, abc.Iterable)`, because `iter(x)` also considers the legacy `__getitem__` method, while the `Iterable` ABC does not.

Explicitly checking whether an object is iterable may not be worthwhile if right after the check you are going to iterate over the object. After all, when the iteration is attempted on a noniterable, the exception Python raises is clear enough: `TypeError: 'C' object is not iterable` . If you can do better than just raising `TypeError`, then do so in a `try/except` block instead of doing an explicit check. The explicit check may make sense if you are holding on to the object to iterate over it later; in this case, catching the error early may be useful.

The next section makes explicit the relationship between iterables and iterators.

Iterables Versus Iterators

From the explanation in "Why Sequences Are Iterable: The iter Function" on page 404 we can extrapolate a definition:

iterable

Any object from which the `iter` built-in function can obtain an iterator. Objects implementing an `__iter__` method returning an *iterator* are iterable. Sequences

WOW! eBook
www.wowebook.org

are always iterable; as are objects implementing a `__getitem__` method that takes 0-based indexes.

It's important to be clear about the relationship between iterables and iterators: Python obtains iterators from iterables.

Here is a simple `for` loop iterating over a `str`. The `str` `'ABC'` is the iterable here. You don't see it, but there is an iterator behind the curtain:

```
>>> s = 'ABC'
>>> for char in s:
...     print(char)
...
A
B
C
```

If there was no `for` statement and we had to emulate the `for` machinery by hand with a `while` loop, this is what we'd have to write:

```
>>> s = 'ABC'
>>> it = iter(s)  # ❶
>>> while True:
...     try:
...         print(next(it))  # ❷
...     except StopIteration:  # ❸
...         del it  # ❹
...         break  # ❺
...
A
B
C
```

❶ Build an iterator `it` from the iterable.

❷ Repeatedly call `next` on the iterator to obtain the next item.

❸ The iterator raises `StopIteration` when there are no further items.

❹ Release reference to `it`—the iterator object is discarded.

❺ Exit the loop.

`StopIteration` signals that the iterator is exhausted. This exception is handled internally in `for` loops and other iteration contexts like list comprehensions, tuple unpacking, etc.

The standard interface for an iterator has two methods:

`__next__`
: Returns the next available item, raising `StopIteration` when there are no more items.

WOW! eBook
www.wowebook.org

`__iter__`
: Returns `self`; this allows iterators to be used where an iterable is expected, for example, in a `for` loop.

This is formalized in the `collections.abc.Iterator` ABC, which defines the `__next__` abstract method, and subclasses `Iterable`—where the abstract `__iter__` method is defined. See Figure 14-1.

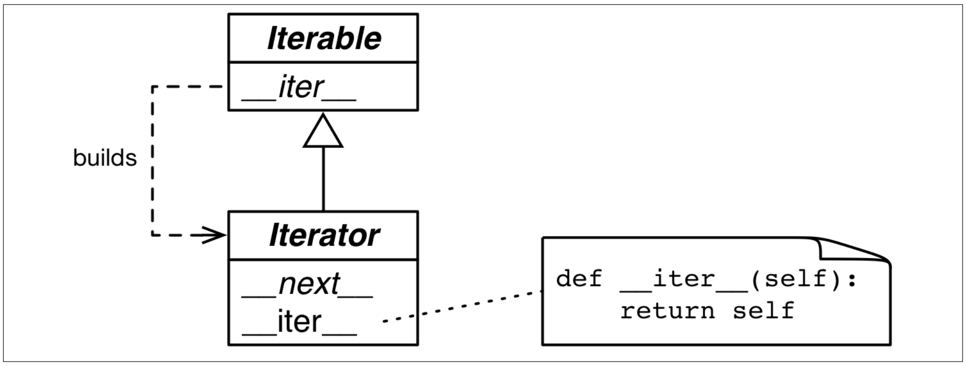

Figure 14-1. The Iterable and Iterator ABCs. Methods in italic are abstract. A concrete Iterable.iter should return a new Iterator instance. A concrete Iterator must implement next. The Iterator.iter method just returns the instance itself.

The `Iterator` ABC implements `__iter__` by doing `return self`. This allows an iterator to be used wherever an iterable is required. The source code for `abc.Iterator` is in Example 14-3.

Example 14-3. abc.Iterator class; extracted from Lib/_collections_abc.py

```python
class Iterator(Iterable):

    __slots__ = ()

    @abstractmethod
    def __next__(self):
        'Return the next item from the iterator. When exhausted, raise StopIteration'
        raise StopIteration

    def __iter__(self):
        return self

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Iterator:
            if (any("__next__" in B.__dict__ for B in C.__mro__) and
                any("__iter__" in B.__dict__ for B in C.__mro__)):
```

WOW! eBook
www.wowebook.org

```
            return True
        return NotImplemented
```

The `Iterator` ABC abstract method is `it.__next__()` in Python 3 and `it.next()` in Python 2. As usual, you should avoid calling special methods directly. Just use the `next(it)`: this built-in function does the right thing in Python 2 and 3.

The *Lib/types.py* module source code in Python 3.4 has a comment that says:

```
# Iterators in Python aren't a matter of type but of protocol.  A large
# and changing number of builtin types implement *some* flavor of
# iterator.  Don't check the type!  Use hasattr to check for both
# "__iter__" and "__next__" attributes instead.
```

In fact, that's exactly what the `__subclasshook__` method of the `abc.Iterator` ABC does (see Example 14-3).

Taking into account the advice from *Lib/types.py* and the logic implemented in *Lib/_collections_abc.py*, the best way to check if an object `x` is an iterator is to call `isinstance(x, abc.Iterator)`. Thanks to `Iterator.__subclasshook__`, this test works even if the class of `x` is not a real or virtual subclass of `Iterator`.

Back to our `Sentence` class from Example 14-1, you can clearly see how the iterator is built by `iter(…)` and consumed by `next(…)` using the Python console:

```
>>> s3 = Sentence('Pig and Pepper')  # ❶
>>> it = iter(s3)  # ❷
>>> it  # doctest: +ELLIPSIS
<iterator object at 0x...>
>>> next(it)  # ❸
'Pig'
>>> next(it)
'and'
>>> next(it)
'Pepper'
>>> next(it)  # ❹
Traceback (most recent call last):
  ...
StopIteration
>>> list(it)  # ❺
[]
>>> list(iter(s3))  # ❻
['Pig', 'and', 'Pepper']
```

❶ Create a sentence `s3` with three words.

WOW! eBook
www.wowebook.org

❷ Obtain an iterator from `s3`.

❸ `next(it)` fetches the next word.

❹ There are no more words, so the iterator raises a `StopIteration` exception.

❺ Once exhausted, an iterator becomes useless.

❻ To go over the sentence again, a new iterator must be built.

Because the only methods required of an iterator are `__next__` and `__iter__`, there is no way to check whether there are remaining items, other than to call `next()` and catch `StopInteration`. Also, it's not possible to "reset" an iterator. If you need to start over, you need to call `iter(…)` on the iterable that built the iterator in the first place. Calling `iter(…)` on the iterator itself won't help, because—as mentioned—`Itera tor.__iter__` is implemented by returning `self`, so this will not reset a depleted iterator.

To wrap up this section, here is a definition for *iterator*:

iterator
: Any object that implements the `__next__` no-argument method that returns the next item in a series or raises `StopIteration` when there are no more items. Python iterators also implement the `__iter__` method so they are *iterable* as well.

This first version of `Sentence` was iterable thanks to the special treatment the `iter(…)` built-in gives to sequences. Now we'll implement the standard iterable protocol.

Sentence Take #2: A Classic Iterator

The next `Sentence` class is built according to the classic Iterator design pattern following the blueprint in the GoF book. Note that this is not idiomatic Python, as the next refactorings will make very clear. But it serves to make explicit the relationship between the iterable collection and the iterator object.

Example 14-4 shows an implementation of a `Sentence` that is iterable because it implements the `__iter__` special method, which builds and returns a `SentenceIterator`. This is how the Iterator design pattern is described in the original *Design Patterns* book.

We are doing it this way here just to make clear the crucial distinction between an iterable and an iterator and how they are connected.

Example 14-4. sentence_iter.py: Sentence implemented using the Iterator pattern

```
import re
import reprlib

RE_WORD = re.compile('\w+')
```

WOW! eBook
www.wowebook.org

```
class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):  ❶
        return SentenceIterator(self.words)  ❷


class SentenceIterator:

    def __init__(self, words):
        self.words = words  ❸
        self.index = 0  ❹

    def __next__(self):
        try:
            word = self.words[self.index]  ❺
        except IndexError:
            raise StopIteration()  ❻
        self.index += 1  ❼
        return word  ❽

    def __iter__(self):  ❾
        return self
```

❶ The `__iter__` method is the only addition to the previous `Sentence` implementation. This version has no `__getitem__`, to make it clear that the class is iterable because it implements `__iter__`.

❷ `__iter__` fulfills the iterable protocol by instantiating and returning an iterator.

❸ `SentenceIterator` holds a reference to the list of words.

❹ `self.index` is used to determine the next word to fetch.

❺ Get the word at `self.index`.

❻ If there is no word at `self.index`, raise `StopIteration`.

❼ Increment `self.index`.

❽ Return the word.

❾ Implement `self.__iter__`.

The code in Example 14-4 passes the tests in Example 14-2.

WOW! eBook
www.wowebook.org

Note that implementing `__iter__` in `SentenceIterator` is not actually needed for this example to work, but the it's the right thing to do: iterators are supposed to implement both `__next__` and `__iter__`, and doing so makes our iterator pass the `issubclass(SentenceInterator, abc.Iterator)` test. If we had subclassed `SentenceIterator` from `abc.Iterator`, we'd inherit the concrete `abc.Iterator.__iter__` method.

That is a lot of work (for us lazy Python programmers, anyway). Note how most code in `SentenceIterator` deals with managing the internal state of the iterator. Soon we'll see how to make it shorter. But first, a brief detour to address an implementation shortcut that may be tempting, but is just wrong.

Making Sentence an Iterator: Bad Idea

A common cause of errors in building iterables and iterators is to confuse the two. To be clear: iterables have an `__iter__` method that instantiates a new iterator every time. Iterators implement a `__next__` method that returns individual items, and an `__iter__` method that returns `self`.

Therefore, iterators are also iterable, but iterables are not iterators.

It may be tempting to implement `__next__` in addition to `__iter__` in the `Sentence` class, making each `Sentence` instance at the same time an iterable and iterator over itself. But this is a terrible idea. It's also a common anti-pattern, according to Alex Martelli who has a lot of experience with Python code reviews.

The "Applicability" section[4] of the Iterator design pattern in the *GoF book* says:

> Use the Iterator pattern
>
> - to access an aggregate object's contents without exposing its internal representation.
> - to support multiple traversals of aggregate objects.
> - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

To "support multiple traversals" it must be possible to obtain multiple independent iterators from the same iterable instance, and each iterator must keep its own internal state, so a proper implementation of the pattern requires each call to `iter(my_iterable)` to create a new, independent, iterator. That is why we need the `SentenceIterator` class in this example.

4. Gamma et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 259.

WOW! eBook
www.wowebook.org

An iterable should never act as an iterator over itself. In other words, iterables must implement `__iter__`, but not `__next__`.

On the other hand, for convenience, iterators should be iterable. An iterator's `__iter__` should just return `self`.

Now that the classic Iterator pattern is properly demonstrated, we can get let it go. The next section presents a more idiomatic implementation of `Sentence`.

Sentence Take #3: A Generator Function

A Pythonic implementation of the same functionality uses a generator function to replace the `SequenceIterator` class. A proper explanation of the generator function comes right after Example 14-5.

Example 14-5. sentence_gen.py: Sentence implemented using a generator function

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for word in self.words:  ❶
            yield word  ❷
        return  ❸

# done! ❹
```

❶ Iterate over `self.word`.

❷ Yield the current `word`.

WOW! eBook
www.wowebook.org

❸ This `return` is not needed; the function can just "fall-through" and return automatically. Either way, a generator function doesn't raise `StopIteration`: it simply exits when it's done producing values.[5]

❹ No need for a separate iterator class!

Here again we have a different implementation of `Sentence` that passes the tests in Example 14-2.

Back in the `Sentence` code in Example 14-4, `__iter__` called the `SentenceIterator` constructor to build an iterator and return it. Now the iterator in Example 14-5 is in fact a generator object, built automatically when the `__iter__` method is called, because `__iter__` here is a generator function.

A full explanation of generator functions follows.

How a Generator Function Works

Any Python function that has the `yield` keyword in its body is a generator function: a function which, when called, returns a generator object. In other words, a generator function is a generator factory.

The only syntax distinguishing a plain function from a generator function is the fact that the latter has a `yield` keyword somewhere in its body. Some argued that a new keyword like `gen` should be used for generator functions instead of `def`, but Guido did not agree. His arguments are in PEP 255 — Simple Generators.[6]

Here is the simplest function useful to demonstrate the behavior of a generator:[7]

```
>>> def gen_123():  # ❶
...     yield 1  # ❷
...     yield 2
...     yield 3
...
>>> gen_123  # doctest: +ELLIPSIS
<function gen_123 at 0x...>  # ❸
```

5. When reviewing this code, Alex Martelli suggested the body of this method could simply be `return iter(self.words)`. He is correct, of course: the result of calling `__iter__` would also be an iterator, as it should be. However, I used a `for` loop with `yield` here to introduce the syntax of a generator function, which will be covered in detail in the next section.

6. Sometimes I add a `gen` prefix or suffix when naming generator functions, but this is not a common practice. And you can't do that if you're implementing an iterable, of course: the necessary special method must be named `__iter__`.

7. Thanks to David Kwast for suggesting this example.

WOW! eBook
www.wowebook.org

```
>>> gen_123()   # doctest: +ELLIPSIS
<generator object gen_123 at 0x...>  # ❹
>>> for i in gen_123():  # ❺
...     print(i)
1
2
3
>>> g = gen_123()  # ❻
>>> next(g)  # ❼
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)  # ❽
Traceback (most recent call last):
  ...
StopIteration
```

❶ Any Python function that contains the `yield` keyword is a generator function.

❷ Usually the body of a generator function has loop, but not necessarily; here I just repeat `yield` three times.

❸ Looking closely, we see `gen_123` is a function object.

❹ But when invoked, `gen_123()` returns a generator object.

❺ Generators are iterators that produce the values of the expressions passed to `yield`.

❻ For closer inspection, we assign the generator object to `g`.

❼ Because `g` is an iterator, calling `next(g)` fetches the next item produced by `yield`.

❽ When the body of the function completes, the generator object raises a `StopIteration`.

A generator function builds a generator object that wraps the body of the function. When we invoke `next(…)` on the generator object, execution advances to the next `yield` in the function body, and the `next(…)` call evaluates to the value yielded when the function body is suspended. Finally, when the function body returns, the enclosing generator object raises `StopIteration`, in accordance with the `Iterator` protocol.

WOW! eBook
www.wowebook.org

I find it helpful to be strict when talking about the results obtained from a generator: I say that a generator *yields* or *produces* values. But it's confusing to say a generator "returns" values. Functions return values. Calling a generator function returns a generator. A generator yields or produces values. A generator doesn't "return" values in the usual way: the `return` statement in the body of a generator function causes `StopIteration` to be raised by the generator object.[8]

Example 14-6 makes the interaction between a `for` loop and the body of the function more explicit.

Example 14-6. A generator function that prints messages when it runs

```
>>> def gen_AB():  # ❶
...     print('start')
...     yield 'A'       # ❷
...     print('continue')
...     yield 'B'       # ❸
...     print('end.')   # ❹
...
>>> for c in gen_AB():  # ❺
...     print('-->', c)  # ❻
...
start     ❼
--> A    ❽
continue ❾
--> B    ❿
end.      ⓫
>>>   ⓬
```

❶ The generator function is defined like any function, but uses `yield`.

❷ The first implicit call to `next()` in the `for` loop at ❺ will print `'start'` and stop at the first `yield`, producing the value `'A'`.

❸ The second implicit call to `next()` in the `for` loop will print `'continue'` and stop at the second `yield`, producing the value `'B'`.

❹ The third call to `next()` will print `'end.'` and fall through the end of the function body, causing the generator object to raise `StopIteration`.

8. Prior to Python 3.3, it was an error to provide a value with the `return` statement in a generator function. Now that is legal, but the `return` still causes a `StopIteration` exception to be raised. The caller can retrieve the return value from the exception object. However, this is only relevant when using a generator function as a coroutine, as we'll see in "Returning a Value from a Coroutine" on page 475.

WOW! eBook
www.wowebook.org

5. To iterate, the for machinery does the equivalent of g = iter(gen_AB()) to get a generator object, and then next(g) at each iteration.
6. The loop block prints --> and the value returned by next(g). But this output will be seen only after the output of the print calls inside the generator function.
7. The string 'start' appears as a result of print('start') in the generator function body.
8. yield 'A' in the generator function body produces the value *A* consumed by the for loop, which gets assigned to the c variable and results in the output --> A.
9. Iteration continues with a second call next(g), advancing the generator function body from yield 'A' to yield 'B'. The text continue is output because of the second print in the generator function body.
10. yield 'B' produces the value *B* consumed by the for loop, which gets assigned to the c loop variable, so the loop prints --> B.
11. Iteration continues with a third call next(it), advancing to the end of the body of the function. The text end. appears in the output because of the third print in the generator function body.
12. When the generator function body runs to the end, the generator object raises StopIteration. The for loop machinery catches that exception, and the loop terminates cleanly.

Now hopefully it's clear how Sentence.__iter__ in Example 14-5 works: __iter__ is a generator function which, when called, builds a generator object that implements the iterator interface, so the SentenceIterator class is no longer needed.

This second version of Sentence is much shorter than the first, but it's not as lazy as it could be. Nowadays, laziness is considered a good trait, at least in programming languages and APIs. A lazy implementation postpones producing values to the last possible moment. This saves memory and may avoid useless processing as well.

We'll build a lazy Sentence class next.

Sentence Take #4: A Lazy Implementation

The Iterator interface is designed to be lazy: next(my_iterator) produces one item at a time. The opposite of lazy is eager: lazy evaluation and eager evaluation are actual technical terms in programming language theory.

Our Sentence implementations so far have not been lazy because the __init__ eagerly builds a list of all words in the text, binding it to the self.words attribute. This will entail processing the entire text, and the list may use as much memory as the text itself

WOW! eBook
www.wowebook.org

(probably more; it depends on how many nonword characters are in the text). Most of this work will be in vain if the user only iterates over the first couple words.

Whenever you are using Python 3 and start wondering “Is there a lazy way of doing this?”, often the answer is “Yes.”

The `re.finditer` function is a lazy version of `re.findall` which, instead of a list, returns a generator producing `re.MatchObject` instances on demand. If there are many matches, `re.finditer` saves a lot of memory. Using it, our third version of `Sentence` is now lazy: it only produces the next word when it is needed. The code is in Example 14-7.

Example 14-7. sentence_gen2.py: Sentence implemented using a generator function calling the re.finditer generator function

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text  ❶

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        for match in RE_WORD.finditer(self.text):  ❷
            yield match.group()  ❸
```

❶ No need to have a `words` list.

❷ `finditer` builds an iterator over the matches of `RE_WORD` on `self.text`, yielding `MatchObject` instances.

❸ `match.group()` extracts the actual matched text from the `MatchObject` instance.

Generator functions are an awesome shortcut, but the code can be made even shorter with a generator expression.

Sentence Take #5: A Generator Expression

Simple generator functions like the one in the previous `Sentence` class (Example 14-7) can be replaced by a generator expression.

A generator expression can be understood as a lazy version of a list comprehension: it does not eagerly build a list, but returns a generator that will lazily produce the items

WOW! eBook
www.wowebook.org

on demand. In other words, if a list comprehension is a factory of lists, a generator expression is a factory of generators.

Example 14-8 is a quick demo of a generator expression, comparing it to a list comprehension.

Example 14-8. The gen_AB generator function is used by a list comprehension, then by a generator expression

```
>>> def gen_AB():  # ❶
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()]  # ❷
start
continue
end.
>>> for i in res1:  # ❸
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB())  # ❹
>>> res2  # ❺
<generator object <genexpr> at 0x10063c240>
>>> for i in res2:  # ❻
...     print('-->', i)
...
start
--> AAA
continue
--> BBB
end.
```

❶ This is the same `gen_AB` function from Example 14-6.

❷ The list comprehension eagerly iterates over the items yielded by the generator object produced by calling `gen_AB()`: `'A'` and `'B'`. Note the output in the next lines: `start`, `continue`, `end.`

❸ This `for` loop is iterating over the `res1` `list` produced by the list comprehension.

❹ The generator expression returns `res2`. The call to `gen_AB()` is made, but that call returns a generator, which is not consumed here.

❺ `res2` is a generator object.

WOW! eBook
www.wowebook.org

❻ Only when the for loop iterates over res2, the body of gen_AB actually executes. Each iteration of the for loop implicitly calls next(res2), advancing gen_AB to the next yield. Note the output of gen_AB with the output of the print in the for loop.

So, a generator expression produces a generator, and we can use it to further reduce the code in the Sentence class. See Example 14-9.

Example 14-9. sentence_genexp.py: Sentence implemented using a generator expression

```
import re
import reprlib

RE_WORD = re.compile('\w+')


class Sentence:

    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return 'Sentence(%s)' % reprlib.repr(self.text)

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

The only difference from Example 14-7 is the __iter__ method, which here is not a generator function (it has no yield) but uses a generator expression to build a generator and then returns it. The end result is the same: the caller of __iter__ gets a generator object.

Generator expressions are syntactic sugar: they can always be replaced by generator functions, but sometimes are more convenient. The next section is about generator expression usage.

Generator Expressions: When to Use Them

I used several generator expressions when implementing the Vector class in Example 10-16. Each of the methods __eq__, __hash__, __abs__, angle, angles, format, __add__, and __mul__ has a generator expression. In all those methods, a list comprehension would also work, at the cost of using more memory to store the intermediate list values.

In Example 14-9, we saw that a generator expression is a syntactic shortcut to create a generator without defining and calling a function. On the other hand, generator func-

WOW! eBook
www.wowebook.org

tions are much more flexible: you can code complex logic with multiple statements, and can even use them as *coroutines* (see Chapter 16).

For the simpler cases, a generator expression will do, and it's easier to read at a glance, as the `Vector` example shows.

My rule of thumb in choosing the syntax to use is simple: if the generator expression spans more than a couple of lines, I prefer to code a generator function for the sake of readability. Also, because generator functions have a name, they can be reused. You can always name a generator expression and use it later by assigning it to a variable, of course, but that is stretching its intended usage as a one-off generator.

Syntax Tip

When a generator expression is passed as the single argument to a function or constructor, you don't need to write a set of parentheses for the function call and another to enclose the generator expression. A single pair will do, like in the `Vector` call from the `__mul__` method in Example 10-16, reproduced here. However, if there are more function arguments after the generator expression, you need to enclose it in parentheses to avoid a `SyntaxError`:

```
def __mul__(self, scalar):
    if isinstance(scalar, numbers.Real):
        return Vector(n * scalar for n in self)
    else:
        return NotImplemented
```

The `Sentence` examples we've seen exemplify the use of generators playing the role of classic iterators: retrieving items from a collection. But generators can also be used to produce values independent of a data source. The next section shows an example of that.

Another Example: Arithmetic Progression Generator

The classic Iterator pattern is all about traversal: navigating some data structure. But a standard interface based on a method to fetch the next item in a series is also useful when the items are produced on the fly, instead of retrieved from a collection. For example, the `range` built-in generates a bounded arithmetic progression (AP) of integers, and the `itertools.count` function generates a boundless AP.

We'll cover `itertools.count` in the next section, but what if you need to generate a bounded AP of numbers of any type?

Example 14-10 shows a few console tests of an `ArithmeticProgression` class we will see in a moment. The signature of the constructor in Example 14-10 is `Arithmetic Progression(begin, step[, end])`. The `range()` function is similar to the `Arithme`

WOW! eBook
www.wowebook.org

ticProgression here, but its full signature is range(start, stop[, step]). I chose to implement a different signature because for an arithmetic progression the step is mandatory but end is optional. I also changed the argument names from start/stop to begin/end to make it very clear that I opted for a different signature. In each test in Example 14-10 I call list() on the result to inspect the generated values.

Example 14-10. Demonstration of an ArithmeticProgression class

```
    >>> ap = ArithmeticProgression(0, 1, 3)
    >>> list(ap)
    [0, 1, 2]
    >>> ap = ArithmeticProgression(1, .5, 3)
    >>> list(ap)
    [1.0, 1.5, 2.0, 2.5]
    >>> ap = ArithmeticProgression(0, 1/3, 1)
    >>> list(ap)
    [0.0, 0.3333333333333333, 0.6666666666666666]
    >>> from fractions import Fraction
    >>> ap = ArithmeticProgression(0, Fraction(1, 3), 1)
    >>> list(ap)
    [Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
    >>> from decimal import Decimal
    >>> ap = ArithmeticProgression(0, Decimal('.1'), .3)
    >>> list(ap)
    [Decimal('0.0'), Decimal('0.1'), Decimal('0.2')]
```

Note that type of the numbers in the resulting arithmetic progression follows the type of begin or step, according to the numeric coercion rules of Python arithmetic. In Example 14-10, you see lists of int, float, Fraction, and Decimal numbers.

Example 14-11 lists the implementation of the ArithmeticProgression class.

Example 14-11. The ArithmeticProgression class

```
class ArithmeticProgression:

    def __init__(self, begin, step, end=None):  ❶
        self.begin = begin
        self.step = step
        self.end = end  # None -> "infinite" series

    def __iter__(self):
        result = type(self.begin + self.step)(self.begin)  ❷
        forever = self.end is None  ❸
        index = 0
        while forever or result < self.end:  ❹
            yield result  ❺
            index += 1
            result = self.begin + self.step * index  ❻
```

WOW! eBook
www.wowebook.org

❶ `__init__` requires two arguments: `begin` and `step`. `end` is optional, if it's `None`, the series will be unbounded.

❷ This line produces a `result` value equal to `self.begin`, but coerced to the type of the subsequent additions.[9]

❸ For readability, the `forever` flag will be `True` if the `self.end` attribute is `None`, resulting in an unbounded series.

❹ This loop runs `forever` or until the result matches or exceeds `self.end`. When this loop exits, so does the function.

❺ The current `result` is produced.

❻ The next potential result is calculated. It may never be yielded, because the `while` loop may terminate.

In the last line of Example 14-11, instead of simply incrementing the `result` with `self.step` iteratively, I opted to use an `index` variable and calculate each `result` by adding `self.begin` to `self.step` multiplied by `index` to reduce the cumulative effect of errors when working with with floats.

The `ArithmeticProgression` class from Example 14-11 works as intended, and is a clear example of the use of a generator function to implement the `__iter__` special method. However, if the whole point of a class is to build a generator by implementing `__iter__`, the class can be reduced to a generator function. A generator function is, after all, a generator factory.

Example 14-12 shows a generator function called `aritprog_gen` that does the same job as `ArithmeticProgression` but with less code. The tests in Example 14-10 all pass if you just call `aritprog_gen` instead of `ArithmeticProgression`.[10]

Example 14-12. The aritprog_gen generator function

```
def aritprog_gen(begin, step, end=None):
    result = type(begin + step)(begin)
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
```

9. In Python 2, there was a `coerce()` built-in function but it's gone in Python 3, deemed unnecessary because the numeric coercion rules are implicit in the arithmetic operator methods. So the best way I could think of to coerce the initial value to be of the same type as the rest of the series was to perform the addition and use its type to convert the result. I asked about this in the Python-list and got an excellent response from Steven D'Aprano.

10. The *14-it-generator/* directory in the *Fluent Python* code repository includes doctests and a script, *aritprog_runner.py*, which runs the tests against all variations of the *aritprog*.py* scripts.

WOW! eBook
www.wowebook.org

```
        index += 1
        result = begin + step * index
```

Example 14-12 is pretty cool, but always remember: there are plenty of ready-to-use generators in the standard library, and the next section will show an even cooler implementation using the `itertools` module.

Arithmetic Progression with itertools

The `itertools` module in Python 3.4 has 19 generator functions that can be combined in a variety of interesting ways.

For example, the `itertools.count` function returns a generator that produces numbers. Without arguments, it produces a series of integers starting with `0`. But you can provide optional `start` and `step` values to achieve a result very similar to our `aritprog_gen` functions:

```
>>> import itertools
>>> gen = itertools.count(1, .5)
>>> next(gen)
1
>>> next(gen)
1.5
>>> next(gen)
2.0
>>> next(gen)
2.5
```

However, `itertools.count` never stops, so if you call `list(count())`, Python will try to build a `list` larger than available memory and your machine will be very grumpy long before the call fails.

On the other hand, there is the `itertools.takewhile` function: it produces a generator that consumes another generator and stops when a given predicate evaluates to `False`. So we can combine the two and write this:

```
>>> gen = itertools.takewhile(lambda n: n < 3, itertools.count(1, .5))
>>> list(gen)
[1, 1.5, 2.0, 2.5]
```

Leveraging `takewhile` and `count`, Example 14-13 is sweet and short.

Example 14-13. aritprog_v3.py: this works like the previous aritprog_gen functions

```
import itertools


def aritprog_gen(begin, step, end=None):
    first = type(begin + step)(begin)
    ap_gen = itertools.count(first, step)
    if end is not None:
```

WOW! eBook
www.wowebook.org

```
        ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
    return ap_gen
```

Note that `aritprog_gen` is not a generator function in Example 14-13: it has no `yield` in its body. But it returns a generator, so it operates as a generator factory, just as a generator function does.

The point of Example 14-13 is: when implementing generators, know what is available in the standard library, otherwise there's a good chance you'll reinvent the wheel. That's why the next section covers several ready-to-use generator functions.

Generator Functions in the Standard Library

The standard library provides many generators, from plain-text file objects providing line-by-line iteration, to the awesome `os.walk` function, which yields filenames while traversing a directory tree, making recursive filesystem searches as simple as a `for` loop.

The `os.walk` generator function is impressive, but in this section I want to focus on general-purpose functions that take arbitrary iterables as arguments and return generators that produce selected, computed, or rearranged items. In the following tables, I summarize two dozen of them, from the built-in, `itertools`, and `functools` modules. For convenience, I grouped them by high-level functionality, regardless of where they are defined.

Perhaps you know all the functions mentioned in this section, but some of them are underused, so a quick overview may be good to recall what's already available.

The first group are filtering generator functions: they yield a subset of items produced by the input iterable, without changing the items themselves. We used `itertools.takewhile` previously in this chapter, in "Arithmetic Progression with itertools" on page 423. Like `takewhile`, most functions listed in Table 14-1 take a `predicate`, which is a one-argument Boolean function that will be applied to each item in the input to determine whether the item is included in the output.

Table 14-1. Filtering generator functions

Module	Function	Description
`itertools`	`compress(it, selector_it)`	Consumes two iterables in parallel; yields items from `it` whenever the corresponding item in `selector_it` is truthy
`itertools`	`dropwhile(predicate, it)`	Consumes `it` skipping items while `predicate` computes truthy, then yields every remaining item (no further checks are made)

WOW! eBook
www.wowebook.org

Module	Function	Description
(built-in)	`filter(predicate, it)`	Applies `predicate` to each item of `iterable`, yielding the item if `predicate(item)` is truthy; if `predicate` is None, only truthy items are yielded
`itertools`	`filterfalse(predicate, it)`	Same as `filter`, with the `predicate` logic negated: yields items whenever `predicate` computes falsy
`itertools`	`islice(it, stop)` or `islice(it, start, stop, step=1)`	Yields items from a slice of `it`, similar to `s[:stop]` or `s[start:stop:step]` except `it` can be any iterable, and the operation is lazy
`itertools`	`takewhile(predicate, it)`	Yields items while `predicate` computes truthy, then stops and no further checks are made

The console listing in Example 14-14 shows the use of all functions in Table 14-1.

Example 14-14. Filtering generator functions examples

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
...
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1,0,1,1,0,1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

The next group are the mapping generators: they yield items computed from each individual item in the input iterable—or iterables, in the case of `map` and `starmap`.[11] The generators in Table 14-2 yield one result per item in the input iterables. If the input comes from more than one iterable, the output stops as soon as the first input iterable is exhausted.

11. Here the term "mapping" is unrelated to dictionaries, but has to do with the `map` built-in.

WOW! eBook
www.wowebook.org

Table 14-2. Mapping generator functions

Module	Function	Description
`itertools`	`accumulate(it, [func])`	Yields accumulated sums; if `func` is provided, yields the result of applying it to the first pair of items, then to the first result and next item, etc.
(built-in)	`enumerate(iterable, start=0)`	Yields 2-tuples of the form `(index, item)`, where `index` is counted from `start`, and `item` is taken from the `iterable`
(built-in)	`map(func, it1, [it2, …, itN])`	Applies `func` to each item of `it`, yielding the result; if N iterables are given, `func` must take N arguments and the iterables will be consumed in parallel
`itertools`	`starmap(func, it)`	Applies `func` to each item of `it`, yielding the result; the input iterable should yield iterable items `iit`, and `func` is applied as `func(*iit)`

Example 14-15 demonstrates some uses of `itertools.accumulate`.

Example 14-15. itertools.accumulate generator function examples

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> import itertools
>>> list(itertools.accumulate(sample))  # ❶
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min))  # ❷
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max))  # ❸
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul))  # ❹
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul))
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]  # ❺
```

❶ Running sum.

❷ Running minimum.

❸ Running maximum.

❹ Running product.

❺ Factorials from `1!` to `10!`.

The remaining functions of Table 14-2 are shown in Example 14-16.

Example 14-16. Mapping generator function examples

```
>>> list(enumerate('albatroz', 1))  # ❶
[(1, 'a'), (2, 'l'), (3, 'b'), (4, 'a'), (5, 't'), (6, 'r'), (7, 'o'), (8, 'z')]
>>> import operator
>>> list(map(operator.mul, range(11), range(11)))  # ❷
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8]))  # ❸
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8]))  # ❹
```

WOW! eBook
www.wowebook.org

```
[(0, 2), (1, 4), (2, 8)]
>>> import itertools
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1)))  # ❺
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b/a,
...     enumerate(itertools.accumulate(sample), 1)))  # ❻
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333,
5.0, 4.375, 4.888888888888889, 4.5]
```

❶ Number the letters in the word, starting from `1`.

❷ Squares of integers from `0` to `10`.

❸ Multiplying numbers from two iterables in parallel: results stop when the shortest iterable ends.

❹ This is what the `zip` built-in function does.

❺ Repeat each letter in the word according to its place in it, starting from `1`.

❻ Running average.

Next, we have the group of merging generators—all of these yield items from multiple input iterables. `chain` and `chain.from_iterable` consume the input iterables sequentially (one after the other), while `product`, `zip`, and `zip_longest` consume the input iterables in parallel. See Table 14-3.

Table 14-3. Generator functions that merge multiple input iterables

Module	Function	Description
`itertools`	`chain(it1, …, itN)`	Yield all items from `it1`, then from `it2` etc., seamlessly
`itertools`	`chain.from_iterable(it)`	Yield all items from each iterable produced by `it`, one after the other, seamlessly; `it` should yield iterable items, for example, a list of iterables
`itertools`	`product(it1, …, itN, repeat=1)`	Cartesian product: yields N-tuples made by combining items from each input iterable like nested `for` loops could produce; `repeat` allows the input iterables to be consumed more than once
(built-in)	`zip(it1, …, itN)`	Yields N-tuples built from items taken from the iterables in parallel, silently stopping when the first iterable is exhausted
`itertools`	`zip_longest(it1, …, itN, fillvalue=None)`	Yields N-tuples built from items taken from the iterables in parallel, stopping only when the last iterable is exhausted, filling the blanks with the `fillvalue`

Example 14-17 shows the use of the `itertools.chain` and `zip` generator functions and their siblings. Recall that the `zip` function is named after the zip fastener or zipper (no relation with compression). Both `zip` and `itertools.zip_longest` were introduced in "The Awesome zip" on page 293.

WOW! eBook
www.wowebook.org

Example 14-17. Merging generator function examples

```
>>> list(itertools.chain('ABC', range(2)))  # ❶
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC')))  # ❷
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC')))  # ❸
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5)))  # ❹
[('A', 0), ('B', 1), ('C', 2)]
>>> list(zip('ABC', range(5), [10, 20, 30, 40]))  # ❺
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5)))  # ❻
[('A', 0), ('B', 1), ('C', 2), (None, 3), (None, 4)]
>>> list(itertools.zip_longest('ABC', range(5), fillvalue='?'))  # ❼
[('A', 0), ('B', 1), ('C', 2), ('?', 3), ('?', 4)]
```

❶ `chain` is usually called with two or more iterables.

❷ `chain` does nothing useful when called with a single iterable.

❸ But `chain.from_iterable` takes each item from the iterable, and chains them in sequence, as long as each item is itself iterable.

❹ `zip` is commonly used to merge two iterables into a series of two-tuples.

❺ Any number of iterables can be consumed by `zip` in parallel, but the generator stops as soon as the first iterable ends.

❻ `itertools.zip_longest` works like `zip`, except it consumes all input iterables to the end, padding output tuples with `None` as needed.

❼ The `fillvalue` keyword argument specifies a custom padding value.

The `itertools.product` generator is a lazy way of computing Cartesian products, which we built using list comprehensions with more than one `for` clause in "Cartesian Products" on page 23. Generator expressions with multiple `for` clauses can also be used to produce Cartesian products lazily. Example 14-18 demonstrates `itertools.product`.

Example 14-18. itertools.product generator function examples

```
>>> list(itertools.product('ABC', range(2)))  # ❶
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'spades hearts diamonds clubs'.split()
>>> list(itertools.product('AK', suits))  # ❷
[('A', 'spades'), ('A', 'hearts'), ('A', 'diamonds'), ('A', 'clubs'),
('K', 'spades'), ('K', 'hearts'), ('K', 'diamonds'), ('K', 'clubs')]
>>> list(itertools.product('ABC'))  # ❸
[('A',), ('B',), ('C',)]
>>> list(itertools.product('ABC', repeat=2))  # ❹
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'),
('B', 'C'), ('C', 'A'), ('C', 'B'), ('C', 'C')]
>>> list(itertools.product(range(2), repeat=3))
```

WOW! eBook
www.wowebook.org

```
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0),
(1, 0, 1), (1, 1, 0), (1, 1, 1)]
>>> rows = itertools.product('AB', range(2), repeat=2)
>>> for row in rows: print(row)
...
('A', 0, 'A', 0)
('A', 0, 'A', 1)
('A', 0, 'B', 0)
('A', 0, 'B', 1)
('A', 1, 'A', 0)
('A', 1, 'A', 1)
('A', 1, 'B', 0)
('A', 1, 'B', 1)
('B', 0, 'A', 0)
('B', 0, 'A', 1)
('B', 0, 'B', 0)
('B', 0, 'B', 1)
('B', 1, 'A', 0)
('B', 1, 'A', 1)
('B', 1, 'B', 0)
('B', 1, 'B', 1)
```

❶ The Cartesian product of a `str` with three characters and a `range` with two integers yields six tuples (because `3 * 2` is `6`).

❷ The product of two card ranks (`'AK'`), and four suits is a series of eight tuples.

❸ Given a single iterable, `product` yields a series of one-tuples, not very useful.

❹ The `repeat=N` keyword argument tells product to consume each input iterable `N` times.

Some generator functions expand the input by yielding more than one value per input item. They are listed in Table 14-4.

Table 14-4. Generator functions that expand each input item into multiple output items

Module	Function	Description
`itertools`	`combinations(it, out_len)`	Yield combinations of `out_len` items from the items yielded by `it`
`itertools`	`combinations_with_replacement(it, out_len)`	Yield combinations of `out_len` items from the items yielded by `it`, including combinations with repeated items
`itertools`	`count(start=0, step=1)`	Yields numbers starting at `start`, incremented by `step`, indefinitely
`itertools`	`cycle(it)`	Yields items from `it` storing a copy of each, then yields the entire sequence repeatedly, indefinitely
`itertools`	`permutations(it, out_len=None)`	Yield permutations of `out_len` items from the items yielded by `it`; by default, `out_len` is `len(list(it))`

WOW! eBook
www.wowebook.org

Module	Function	Description
`itertools`	`repeat(item, [times])`	Yield the given item repeadedly, indefinetly unless a number of `times` is given

The `count` and `repeat` functions from `itertools` return generators that conjure items out of nothing: neither of them takes an iterable as input. We saw `itertools.count` in "Arithmetic Progression with itertools" on page 423. The `cycle` generator makes a backup of the input iterable and yields its items repeatedly. Example 14-19 illustrates the use of `count`, `repeat`, and `cycle`.

Example 14-19. count, cycle, and repeat

```
>>> ct = itertools.count()  # ❶
>>> next(ct)  # ❷
0
>>> next(ct), next(ct), next(ct)  # ❸
(1, 2, 3)
>>> list(itertools.islice(itertools.count(1, .3), 3))  # ❹
[1, 1.3, 1.6]
>>> cy = itertools.cycle('ABC')  # ❺
>>> next(cy)
'A'
>>> list(itertools.islice(cy, 7))  # ❻
['B', 'C', 'A', 'B', 'C', 'A', 'B']
>>> rp = itertools.repeat(7)  # ❼
>>> next(rp), next(rp)
(7, 7)
>>> list(itertools.repeat(8, 4))  # ❽
[8, 8, 8, 8]
>>> list(map(operator.mul, range(11), itertools.repeat(5)))  # ❾
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

❶ Build a `count` generator `ct`.

❷ Retrieve the first item from `ct`.

❸ I can't build a `list` from `ct`, because `ct` never stops, so I fetch the next three items.

❹ I can build a `list` from a `count` generator if it is limited by `islice` or `takewhile`.

❺ Build a `cycle` generator from `'ABC'` and fetch its first item, `'A'`.

❻ A `list` can only be built if limited by `islice`; the next seven items are retrieved here.

❼ Build a `repeat` generator that will yield the number 7 forever.

❽ A `repeat` generator can be limited by passing the `times` argument: here the number 8 will be produced 4 times.

WOW! eBook
www.wowebook.org

9. A common use of `repeat`: providing a fixed argument in `map`; here it provides the 5 multiplier.

The `combinations`, `combinations_with_replacement`, and `permutations` generator functions—together with `product`—are called the *combinatoric generators* in the `itertools` documentation page. There is a close relationship between `itertools.product` and the remaining *combinatoric* functions as well, as Example 14-20 shows.

Example 14-20. Combinatoric generator functions yield multiple values per input item

```
>>> list(itertools.combinations('ABC', 2))  # 1
[('A', 'B'), ('A', 'C'), ('B', 'C')]
>>> list(itertools.combinations_with_replacement('ABC', 2))  # 2
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
>>> list(itertools.permutations('ABC', 2))  # 3
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
>>> list(itertools.product('ABC', repeat=2))  # 4
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'B'), ('B', 'C'),
('C', 'A'), ('C', 'B'), ('C', 'C')]
```

1. All combinations of `len()==2` from the items in `'ABC'`; item ordering in the generated tuples is irrelevant (they could be sets).
2. All combinations of `len()==2` from the items in `'ABC'`, including combinations with repeated items.
3. All permutations of `len()==2` from the items in `'ABC'`; item ordering in the generated tuples is relevant.
4. Cartesian product from `'ABC'` and `'ABC'` (that's the effect of `repeat=2`).

The last group of generator functions we'll cover in this section are designed to yield all items in the input iterables, but rearranged in some way. Here are two functions that return multiple generators: `itertools.groupby` and `itertools.tee`. The other generator function in this group, the `reversed` built-in, is the only one covered in this section that does not accept any iterable as input, but only sequences. This makes sense: because `reversed` will yield the items from last to first, it only works with a sequence with a known length. But it avoids the cost of making a reversed copy of the sequence by yielding each item as needed. I put the `itertools.product` function together with the *merging* generators in Table 14-3 because they all consume more than one iterable, while the generators in Table 14-5 all accept at most one input iterable.

Table 14-5. Rearranging generator functions

Module	Function	Description
`itertools`	`groupby(it, key=None)`	Yields 2-tuples of the form `(key, group)`, where `key` is the grouping criterion and `group` is a generator yielding the items in the group

WOW! eBook
www.wowebook.org

Module	Function	Description
(built-in)	`reversed(seq)`	Yields items from `seq` in reverse order, from last to first; `seq` must be a sequence or implement the `__reversed__` special method
`itertools`	`tee(it, n=2)`	Yields a tuple of *n* generators, each yielding the items of the input iterable independently

Example 14-21 demonstrates the use of `itertools.groupby` and the `reversed` built-in. Note that `itertools.groupby` assumes that the input iterable is sorted by the grouping criterion, or at least that the items are clustered by that criterion—even if not sorted.

Example 14-21. itertools.groupby

```
>>> list(itertools.groupby('LLLLAAGGG'))  # ❶
[('L', <itertools._grouper object at 0x102227cc0>),
('A', <itertools._grouper object at 0x102227b38>),
('G', <itertools._grouper object at 0x102227b70>)]
>>> for char, group in itertools.groupby('LLLLAAAGG'):  # ❷
...     print(char, '->', list(group))
...
L -> ['L', 'L', 'L', 'L']
A -> ['A', 'A',]
G -> ['G', 'G', 'G']
>>> animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
...            'bat', 'dolphin', 'shark', 'lion']
>>> animals.sort(key=len)  # ❸
>>> animals
['rat', 'bat', 'duck', 'bear', 'lion', 'eagle', 'shark',
'giraffe', 'dolphin']
>>> for length, group in itertools.groupby(animals, len):  # ❹
...     print(length, '->', list(group))
...
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
>>> for length, group in itertools.groupby(reversed(animals), len): # ❺
...     print(length, '->', list(group))
...
7 -> ['dolphin', 'giraffe']
5 -> ['shark', 'eagle']
4 -> ['lion', 'bear', 'duck']
3 -> ['bat', 'rat']
>>>
```

❶ `groupby` yields tuples of `(key, group_generator)`.

❷ Handling `groupby` generators involves nested iteration: in this case, the outer `for` loop and the inner `list` constructor.

❸ To use `groupby`, the input should be sorted; here the words are sorted by length.

WOW! eBook
www.wowebook.org

❹ Again, loop over the `key` and `group` pair, to display the `key` and expand the `group` into a `list`.

❺ Here the `reverse` generator is used to iterate over `animals` from right to left.

The last of the generator functions in this group is `iterator.tee`, which has a unique behavior: it yields multiple generators from a single input iterable, each yielding every item from the input. Those generators can be consumed independently, as shown in Example 14-22.

Example 14-22. itertools.tee yields multiple generators, each yielding every item of the input generator

```
>>> list(itertools.tee('ABC'))
[<itertools._tee object at 0x10222abc8>, <itertools._tee object at 0x10222ac08>]
>>> g1, g2 = itertools.tee('ABC')
>>> next(g1)
'A'
>>> next(g2)
'A'
>>> next(g2)
'B'
>>> list(g1)
['B', 'C']
>>> list(g2)
['C']
>>> list(zip(*itertools.tee('ABC')))
[('A', 'A'), ('B', 'B'), ('C', 'C')]
```

Note that several examples in this section used combinations of generator functions. This is a great feature of these functions: because they all take generators as arguments and return generators, they can be combined in many different ways.

While on the subject of combining generators, the `yield from` statement, new in Python 3.3, is a tool for doing just that.

New Syntax in Python 3.3: yield from

Nested for loops are the traditional solution when a generator function needs to yield values produced from another generator.

For example, here is a homemade implementation of a chaining generator:[12]

```
    >>> def chain(*iterables):
    ...     for it in iterables:
    ...         for i in it:
```

12. The `itertools.chain` from the standard library is written in C.

WOW! eBook
www.wowebook.org

```
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

The `chain` generator function is delegating to each received iterable in turn. PEP 380 — Syntax for Delegating to a Subgenerator introduced new syntax for doing that, shown in the next console listing:

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i
...
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

As you can see, `yield from i` replaces the inner `for` loop completely. The use of `yield from` in this example is correct, and the code reads better, but it seems like mere syntactic sugar. Besides replacing a loop, `yield from` creates a channel connecting the inner generator directly to the client of the outer generator. This channel becomes really important when generators are used as coroutines and not only produce but also consume values from the client code. Chapter 16 dives into coroutines, and has several pages explaining why `yield from` is much more than syntactic sugar.

After this first encounter with `yield from`, we'll go back to our review of iterable-savvy functions in the standard library.

Iterable Reducing Functions

The functions in Table 14-6 all take an iterable and return a single result. They are known as "reducing," "folding," or "accumulating" functions. Actually, every one of the built-ins listed here can be implemented with `functools.reduce`, but they exist as built-ins because they address some common use cases more easily. Also, in the case of `all` and `any`, there is an important optimization that can't be done with `reduce`: these functions short-circuit (i.e., they stop consuming the iterator as soon as the result is determined). See the last test with `any` in Example 14-23.

Table 14-6. Built-in functions that read iterables and return single values

Module	Function	Description
(built-in)	`all(it)`	Returns `True` if all items in `it` are truthy, otherwise `False`; `all([])` returns `True`
(built-in)	`any(it)`	Returns `True` if any item in `it` is truthy, otherwise `False`; `any([])` returns `False`

WOW! eBook
www.wowebook.org

Module	Function	Description
(built-in)	`max(it, [key=,] [de fault=])`	Returns the maximum value of the items in `it`;[a] `key` is an ordering function, as in `sorted`; `default` is returned if the iterable is empty
(built-in)	`min(it, [key=,] [de fault=])`	Returns the minimum value of the items in `it`.[b] `key` is an ordering function, as in `sorted`; `default` is returned if the iterable is empty
`functools`	`reduce(func, it, [ini tial])`	Returns the result of applying `func` to the first pair of items, then to that result and the third item and so on; if given, `initial` forms the initial pair with the first item
(built-in)	`sum(it, start=0)`	The sum of all items in `it`, with the optional `start` value added (use `math.fsum` for better precision when adding floats)

[a] May also be called as `max(arg1, arg2, …, [key=?])`, in which case the maximum among the arguments is returned.

[b] May also be called as `min(arg1, arg2, …, [key=?])`, in which case the minimum among the arguments is returned.

The operation of `all` and `any` is exemplified in Example 14-23.

Example 14-23. Results of all and any for some sequences

```
>>> all([1, 2, 3])
True
>>> all([1, 0, 3])
False
>>> all([])
True
>>> any([1, 2, 3])
True
>>> any([1, 0, 3])
True
>>> any([0, 0.0])
False
>>> any([])
False
>>> g = (n for n in [0, 0.0, 7, 8])
>>> any(g)
True
>>> next(g)
8
```

A longer explanation about `functools.reduce` appeared in "Vector Take #4: Hashing and a Faster ==" on page 288.

Another built-in that takes an iterable and returns something else is `sorted`. Unlike `reversed`, which is a generator function, `sorted` builds and returns an actual list. After all, every single item of the input iterable must be read so they can be sorted, and the sorting happens in a `list`, therefore `sorted` just returns that `list` after it's done. I mention `sorted` here because it does consume an arbitrary iterable.

WOW! eBook
www.wowebook.org

Of course, `sorted` and the reducing functions only work with iterables that eventually stop. Otherwise, they will keep on collecting items and never return a result.

We'll now go back to the `iter()` built-in: it has a little-known feature that we haven't covered yet.

A Closer Look at the iter Function

As we've seen, Python calls `iter(x)` when it needs to iterate over an object `x`.

But `iter` has another trick: it can be called with two arguments to create an iterator from a regular function or any callable object. In this usage, the first argument must be a callable to be invoked repeatedly (with no arguments) to yield values, and the second argument is a sentinel: a marker value which, when returned by the callable, causes the iterator to raise `StopIteration` instead of yielding the sentinel.

The following example shows how to use `iter` to roll a six-sided die until a `1` is rolled:

```
>>> def d6():
...     return randint(1, 6)
...
>>> d6_iter = iter(d6, 1)
>>> d6_iter
<callable_iterator object at 0x00000000029BE6A0>
>>> for roll in d6_iter:
...     print(roll)
...
4
3
6
3
```

Note that the `iter` function here returns a `callable_iterator`. The `for` loop in the example may run for a very long time, but it will never display `1`, because that is the sentinel value. As usual with iterators, the `d6_iter` object in the example becomes useless once exhausted. To start over, you must rebuild the iterator by invoking `iter(…)` again.

A useful example is found in the `iter` built-in function documentation. This snippet reads lines from a file until a blank line is found or the end of file is reached:

```
with open('mydata.txt') as fp:
    for line in iter(fp.readline, ''):
        process_line(line)
```

To close this chapter, I present a practical example of using generators to handle a large volume of data efficiently.

WOW! eBook
www.wowebook.org

Case Study: Generators in a Database Conversion Utility

A few years ago I worked at BIREME, a digital library run by PAHO/WHO (Pan-American Health Organization/World Health Organization) in São Paulo, Brazil. Among the bibliographic datasets created by BIREME are LILACS (Latin American and Caribbean Health Sciences index) and SciELO (Scientific Electronic Library Online), two comprehensive databases indexing the scientific and technical literature produced in the region.

Since the late 1980s, the database system used to manage LILACS is CDS/ISIS, a non-relational, document database created by UNESCO and eventually rewritten in C by BIREME to run on GNU/Linux servers. One of my jobs was to research alternatives for a possible migration of LILACS—and eventually the much larger SciELO—to a modern, open source, document database such as CouchDB or MongoDB.

As part of that research, I wrote a Python script, *isis2json.py*, that reads a CDS/ISIS file and writes a JSON file suitable for importing to CouchDB or MongoDB. Initially, the script read files in the ISO-2709 format exported by CDS/ISIS. The reading and writing had to be done incrementally because the full datasets were much bigger than main memory. That was easy enough: each iteration of the main `for` loop read one record from the *.iso* file, massaged it, and wrote it to the *.json* output.

However, for operational reasons, it was deemed necessary that *isis2json.py* supported another CDS/ISIS data format: the binary *.mst* files used in production at BIREME—to avoid the costly export to ISO-2709.

Now I had a problem: the libraries used to read ISO-2709 and *.mst* files had very different APIs. And the JSON writing loop was already complicated because the script accepted a variety of command-line options to restructure each output record. Reading data using two different APIs in the same `for` loop where the JSON was produced would be unwieldy.

The solution was to isolate the reading logic into a pair of generator functions: one for each supported input format. In the end, the *isis2json.py* script was split into four functions. You can see the main Python 2 script in Example A-5, but the full source code with dependencies is in *fluentpython/isis2json* on GitHub.

Here is a high-level overview of how the script is structured:

`main`
: The `main` function uses `argparse` to read command-line options that configure the structure of the output records. Based on the input filename extension, a suitable generator function is selected to read the data and yield the records, one by one.

WOW! eBook
www.wowebook.org

`iter_iso_records`
: This generator function reads *.iso* files (assumed to be in the ISO-2709 format). It takes two arguments: the filename and `isis_json_type`, one of the options related to the record structure. Each iteration of its `for` loop reads one record, creates an empty `dict`, populates it with field data, and yields the `dict`.

`iter_mst_records`
: This other generator functions reads *.mst* files.[13] If you look at the source code for *isis2json.py*, you'll see that it's not as simple as `iter_iso_records`, but its interface and overall structure is the same: it takes a filename and an `isis_json_type` argument and enters a `for` loop, which builds and yields one `dict` per iteration, representing a single record.

`write_json`
: This function performs the actual writing of the JSON records, one at a time. It takes numerous arguments, but the first one—`input_gen`—is a reference to a generator function: either `iter_iso_records` or `iter_mst_records`. The main `for` loop in `write_json` iterates over the dictionaries yielded by the selected `input_gen` generator, massages it in several ways as determined by the command-line options, and appends the JSON record to the output file.

By leveraging generator functions, I was able to decouple the reading logic from the writing logic. Of course, the simplest way to decouple them would be to read all records to memory, then write them to disk. But that was not a viable option because of the size of the datasets. Using generators, the reading and writing is interleaved, so the script can process files of any size.

Now if *isis2json.py* needs to support an additional input format—say, MARCXML, a DTD used by the U.S. Library of Congress to represent ISO-2709 data—it will be easy to add a third generator function to implement the reading logic, without changing anything in the complicated `write_json` function.

This is not rocket science, but it's a real example where generators provided a flexible solution to processing databases as a stream of records, keeping memory usage low regardless of the amount of data. Anyone who manages large datasets finds many opportunities for using generators in practice.

The next section addresses an aspect of generators that we'll actually skip for now. Read on to understand why.

13. The library used to read the complex *.mst* binary is actually written in Java, so this functionality is only available when *isis2json.py* is executed with the Jython interpreter, version 2.5 or newer. For further details, see the *README.rst* file in the repository. The dependencies are imported inside the generator functions that need them, so the script can run even if only one of the external libraries is available.

WOW! eBook
www.wowebook.org

Generators as Coroutines

About five years after generator functions with the `yield` keyword were introduced in Python 2.2, PEP 342 — Coroutines via Enhanced Generators was implemented in Python 2.5. This proposal added extra methods and functionality to generator objects, most notably the `.send()` method.

Like `.__next__()`, `.send()` causes the generator to advance to the next `yield`, but it also allows the client using the generator to send data into it: whatever argument is passed to `.send()` becomes the value of the corresponding `yield` expression inside the generator function body. In other words, `.send()` allows two-way data exchange between the client code and the generator—in contrast with `.__next__()`, which only lets the client receive data from the generator.

This is such a major "enhancement" that it actually changes the nature of generators: when used in this way, they become *coroutines*. David Beazley—probably the most prolific writer and speaker about coroutines in the Python community—warned in a famous PyCon US 2009 tutorial:

> - Generators produce data for iteration
> - Coroutines are consumers of data
> - To keep your brain from exploding, you don't mix the two concepts together
> - Coroutines are not related to iteration
> - Note: There is a use of having yield produce a value in a coroutine, but it's not tied to iteration.[14]
>
> — David Beazley
> *"A Curious Course on Coroutines and Concurrency"*

I will follow Dave's advice and close this chapter—which is really about iteration techniques—without touching `send` and the other features that make generators usable as coroutines. Coroutines will be covered in Chapter 16.

Chapter Summary

Iteration is so deeply embedded in the language that I like to say that Python groks iterators.[15] The integration of the Iterator pattern in the semantics of Python is a prime example of how design patterns are not equally applicable in all programming lan-

14. Slide 33, "Keeping It Straight," in "A Curious Course on Coroutines and Concurrency".

15. According to the Jargon file, to *grok* is not merely to learn something, but to absorb it so "it becomes part of you, part of your identity."

WOW! eBook
www.wowebook.org

guages. In Python, a classic iterator implemented "by hand" as in Example 14-4 has no practical use, except as a didactic example.

In this chapter, we built a few versions of a class to iterate over individual words in text files that may be very long. Thanks to the use of generators, the successive refactorings of the `Sentence` class become shorter and easier to read—when you know how they work.

We then coded a generator of arithmetic progressions and showed how to leverage the `itertools` module to make it simpler. An overview of 24 general-purpose generator functions in the standard library followed.

Following that, we looked at the `iter` built-in function: first, to see how it returns an iterator when called as `iter(o)`, and then to study how it builds an iterator from any function when called as `iter(func, sentinel)`.

For practical context, I described the implementation of a database conversion utility using generator functions to decouple the reading to the writing logic, enabling efficient handling of large datasets and making it easy to support more than one data input format.

Also mentioned in this chapter were the `yield from` syntax, new in Python 3.3, and coroutines. Both topics were just introduced here; they get more coverage later in the book.

Further Reading

A detailed technical explanation of generators appears in The Python Language Reference in 6.2.9. Yield expressions. The PEP where generator functions were defined is PEP 255 — Simple Generators.

The `itertools` module documentation is excellent because of all the examples included. Although the functions in that module are implemented in C, the documentation shows how many of them would be written in Python, often by leveraging other functions in the module. The usage examples are also great: for instance, there is a snippet showing how to use the `accumulate` function to amortize a loan with interest, given a list of payments over time. There is also an Itertools Recipes section with additional high-performance functions that use the `itertools` functions as building blocks.

Chapter 4, "Iterators and Generators," of *Python Cookbook, 3E* (O'Reilly), by David Beazley and Brian K. Jones, has 16 recipes covering this subject from many different angles, always focusing on practical applications.

The `yield from` syntax is explained with examples in What's New in Python 3.3 (see PEP 380: Syntax for Delegating to a Subgenerator). We'll also cover it in detail in "Using yield from" on page 477 and "The Meaning of yield from" on page 483 in Chapter 16.

WOW! eBook
www.wowebook.org

If you are interested in document databases and would like to learn more about the context of "Case Study: Generators in a Database Conversion Utility" on page 437, the Code4Lib Journal—which covers the intersection between libraries and technology—published my paper "From ISIS to CouchDB: Databases and Data Models for Bibliographic Records". One section of the paper describes the *isis2json.py* script. The rest of it explains why and how the semistructured data model implemented by document databases like CouchDB and MongoDB are more suitable for cooperative bibliographic data collection than the relational model.

Soapbox

Generator Function Syntax: More Sugar Would Be Nice

> Designers need to ensure that controls and displays for different purposes are significantly different from one another.
>
> — Donald Norman
> *The Design of Everyday Things*

Source code plays the role of "controls and displays" in programming languages. I think Python is exceptionally well designed; its source code is often as readable as pseudocode. But nothing is perfect. Guido van Rossum should have followed Donald Norman's advice (previously quoted) and introduced another keyword for defining generator expressions, instead of reusing `def`. The "BDFL Pronouncements" section of PEP 255 — Simple Generators actually argues:

> A "yield" statement buried in the body is not enough warning that the semantics are so different.

But Guido hates introducing new keywords and he did not find that argument convincing, so we are stuck with `def`.

Reusing the function syntax for generators has other bad consequences. In the paper and experimental work "Python, the Full Monty: A Tested Semantics for the Python Programming Language," Politz[16] et al. show this trivial example of a generator function (section 4.1 of the paper):

```
def f(): x=0
    while True:
        x += 1
        yield x
```

The authors then make the point that we can't abstract the process of yielding with a function call (Example 14-24).

16. Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi, "Python: The Full Monty," SIGPLAN Not. 48, 10 (October 2013), 217-232.

WOW! eBook
www.wowebook.org

Example 14-24. “[This] seems to perform a simple abstraction over the process of yielding” (Politz et al.)

```
def f():
    def do_yield(n):
        yield n
    x = 0
    while True:
        x += 1
        do_yield(x)
```

If we call `f()` in Example 14-24, we get an infinite loop, and not a generator, because the `yield` keyword only makes the immediately enclosing function a generator function. Although generator functions look like functions, we cannot delegate another generator function with a simple function call. As a point of comparison, the Lua language does not impose this limitation. A Lua coroutine can call other functions and any of them can yield to the original caller.

The new `yield from` syntax was introduced to allow a Python generator or coroutine to delegate work to another, without requiring the workaround of an inner `for` loop. Example 14-24 can be “fixed” by prefixing the function call with `yield from`, as in Example 14-25.

Example 14-25. This actually performs a simple abstraction over the process of yielding

```
def f():
    def do_yield(n):
        yield n
    x = 0
    while True:
        x += 1
        yield from do_yield(x)
```

Reusing `def` for declaring generators was a usability mistake, and the problem was compounded in Python 2.5 with coroutines, which are also coded as functions with `yield`. In the case of coroutines, the `yield` just happens to appear—usually—on the righthand side of an assignment, because it receives the argument of the `.send()` call from the client. As David Beazley says:

> Despite some similarities, generators and coroutines are basically two different concepts.[17]

I believe coroutines also deserved their own keyword. As we’ll see later, coroutines are often used with special decorators, which do set them apart from other functions. But generator functions are not decorated as frequently, so we have to scan their bodies for `yield` to realize they are not functions at all, but a completely different beast.

17. Slide 31, “A Curious Course on Coroutines and Concurrency”.

WOW! eBook
www.wowebook.org

It can be argued that, because those features were made to work with little additional syntax, extra syntax would be merely "syntactic sugar." I happen to like syntactic sugar when it makes features that are different look different. The lack of syntactic sugar is the main reason why Lisp code is hard to read: every language construct in Lisp looks like a function call.

Semantics of Generator Versus Iterator

There are at least three ways of thinking about the relationship between iterators and generators.

The first is the interface viewpoint. The Python iterator protocol defines two methods: `__next__` and `__iter__`. Generator objects implement both, so from this perspective, every generator is an iterator. By this definition, objects created by the `enumerate()` built-in are iterators:

```
>>> from collections import abc
>>> e = enumerate('ABC')
>>> isinstance(e, abc.Iterator)
True
```

The second is the implementation viewpoint. From this angle, a generator is a Python language construct that can be coded in two ways: as a function with the `yield` keyword or as a generator expression. The generator objects resulting from calling a generator function or evaluating a generator expression are instances of an internal `GeneratorType`. From this perspective, every generator is also an iterator, because `GeneratorType` instances implement the iterator interface. But you can write an iterator that is not a generator—by implementing the classic Iterator pattern, as we saw in Example 14-4, or by coding an extension in C. The `enumerate` objects are not generators from this perspective:

```
>>> import types
>>> e = enumerate('ABC')
>>> isinstance(e, types.GeneratorType)
False
```

This happens because `types.GeneratorType` is defined as "The type of generator-iterator objects, produced by calling a generator function."

The third is the conceptual viewpoint. In the classic Iterator design pattern—as defined in the GoF book—the iterator traverses a collection and yields items from it. The iterator may be quite complex; for example, it may navigate through a tree-like data structure. But, however much logic is in a classic iterator, it always reads values from an existing data source, and when you call `next(it)`, the iterator is not expected to change the item it gets from the source; it's supposed to just yield it as is.

In contrast, a generator may produce values without necessarily traversing a collection, like `range` does. And even if attached to a collection, generators are not limited to yielding just the items in it, but may yield some other values derived from them. A clear example of this is the `enumerate` function. By the original definition of the design pat-

WOW! eBook
www.wowebook.org

tern, the generator returned by `enumerate` is not an iterator because it creates the tuples it yields.

At this conceptual level, the implementation technique is irrelevant. You can write a generator without using a Python generator object. Example 14-26 is a Fibonacci generator I wrote just to make this point.

Example 14-26. fibo_by_hand.py: Fibonacci generator without GeneratorType instances

```
class Fibonacci:

    def __iter__(self):
        return FibonacciGenerator()


class FibonacciGenerator:

    def __init__(self):
        self.a = 0
        self.b = 1

    def __next__(self):
        result = self.a
        self.a, self.b = self.b, self.a + self.b
        return result

    def __iter__(self):
        return self
```

Example 14-26 works but is just a silly example. Here is the Pythonic Fibonacci generator:

```
    def fibonacci():
        a, b = 0, 1
        while True:
            yield a
            a, b = b, a + b
```

And of course, you can always use the generator language construct to perform the basic duties of an iterator: traversing a collection and yielding items from it.

In reality, Python programmers are not strict about this distinction: generators are also called iterators, even in the official docs. The canonical definition of an iterator in the Python Glossary is so general it encompasses both iterators and generators:

> Iterator: An object representing a stream of data. […]

The full definition of *iterator* in the Python Glossary is worth reading. On the other hand, the definition of *generator* there treats *iterator* and *generator* as synonyms, and uses the word "generator" to refer both to the generator function and the generator

WOW! eBook
www.wowebook.org

object it builds. So, in the Python community lingo, iterator and generator are fairly close synonyms.

The Minimalistic Iterator Interface in Python

In the "Implementation" section of the Iterator pattern,[18] the *Gang of Four* wrote:

> The minimal interface to Iterator consists of the operations First, Next, IsDone, and CurrentItem.

However, that very sentence has a footnote which reads:

> We can make this interface even smaller by merging Next, IsDone, and CurrentItem into a single operation that advances to the next object and returns it. If the traversal is finished, then this operation returns a special value (0, for instance) that marks the end of the iteration.

This is close to what we have in Python: the single method `__next__` does the job. But instead of using a sentinel, which could be overlooked by mistake, the `StopIteration` exception signals the end of the iteration. Simple and correct: that's the Python way.

18. Gamma et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, p. 261.

WOW! eBook
www.wowebook.org