Scaling Python With Generators

Aaron Maxwell

powerfulpython.com

Contents

Chapter 1

Scaling With Generators

This `for` loop seems simple:

```python
for item in items:
    do_something_with(item)
```

And yet, miracles hide here. As you probably know, the act of efficiently going through a collection, one element at a time, is called *iteration*. But few understand how Python's iteration system really works… how deep and well-thought-out it is. This chapter makes you one of those people, giving you the ability to naturally write **highly scalable** Python applications… able to handle ever-larger data sets in performant, memory-efficient ways.

Iteration is also core to one of Python's most powerful tools: the *generator function*. Generator functions are not just a convenient way to create useful iterators. They enable some exquisite patterns of code organization, in a way that - by their very nature - intrinsically encourage excellent coding habits.

This chapter is special, because understanding it threatens to make you a permanently better programmer *in every language*. Mastering Python generators tends to do that, because of the distinctions and insights you gain along the way. Let's dive in.

1.1 Iteration in Python

Python has a built-in function called `iter()`. When you pass it a collection, you get back an *iterator object*:

```
>>> numbers = [7, 4, 11, 3]
>>> iter(numbers)
<list_iterator object at 0x10219dc50>
```

Just as in other languages, a Python iterator produces the values in a sequence, one at a time. You probably know an iterator is like a moving pointer over the collection:

```
>>> numbers_iter = iter(numbers)
>>> for num in numbers_iter: print(num)
7
4
11
3
```

You don't normally need to do this. If you instead write `for num in numbers`, what Python effectively does under the hood is call `iter()` on that collection. This happens automatically. Whatever object it gets back is used as the iterator for that `for` loop:

```
# This...
for num in numbers:
    print(num)

# ... is effectively just like this:
numbers_iter = iter(numbers)
for num in numbers_iter:
    print(num)
```

An iterator over a collection is a separate object, with its own identity - which you can verify with `id()`:

```
>>> # id() returns a unique number for each object.
... # Different objects will always have different IDs.
>>> id(numbers)
4330133896
>>> id(numbers_iter)
4330216640
```

How does `iter()` actually get the iterator? It can do this in several ways, but one relies on a magic method called `__iter__`. This is a method any class (including yours) may define; when called with no arguments, it must return a fresh iterator object. Lists have it, for example:

```
>>> numbers.__iter__
<method-wrapper '__iter__' of list object at 0x10130e4c8>
>>> numbers.__iter__()
<list_iterator object at 0x1013180f0>
```

Python makes a distinction between objects which are *iterators*, and objects which are *iterable*. We say an object is *iterable* if and only if you can pass it to `iter()`, and get a ready-to-use iterator. If that object has an `__iter__` method, `iter()` will call it to get the iterator. Python lists and tuples are iterable. So are strings, which is why you can write `for char in my_str:` to iterate over `my_str`'s characters. Any container you might use in a `for` loop is iterable.

A `for` loop is the most common way to step through a sequence. But sometimes your code needs to step through in a more fine-grained way. For this, use the built-in function `next()`. You normally call it with a single argument, which is an iterator. Each time you call it, `next(my_iterator)` fetches and returns the next element:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> # Create a fresh iterator...
... names_it = iter(names)
>>> next(names_it)
'Tom'
>>> next(names_it)
'Shelly'
>>> next(names_it)
'Garth'
```

What happens if you call `next(names_it)` again? `next()` will raise a special built-in exception, called `StopIteration`:

```
>>> next(names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

This is part of Python's *iterator protocol*. Raising this specific exception is, by design, how an iterator signals the sequence is done. You rarely have to raise or catch this exception yourself, though we'll see some patterns later where it's useful to do so. A good mental model for how a `for` loop works is to imagine it calling `next()` each time through the loop, exiting when `StopIteration` gets raised.

When using `next()` yourself, you can provide a second argument, for the default value. If you do, `next()` will return that instead of raising `StopIteration` at the end:

```
>>> names = ["Tom", "Shelly", "Garth"]
>>> new_names_it = iter(names)
>>> next(new_names_it, "Rick")
'Tom'
>>> next(new_names_it, "Rick")
'Shelly'
>>> next(new_names_it, "Rick")
'Garth'
>>> next(new_names_it, "Rick")
'Rick'
>>> next(new_names_it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(new_names_it, "Jane")
'Jane'
```

Now, let's consider a different situation. What if you aren't working with a simple sequence of numbers or strings, but something more complex? What if you are calculating or reading or otherwise obtaining the sequence elements as you go along? Let's start with a simple example (so it's easy to reason about). Suppose you need to write a function creating a list of square numbers, which will be processed by other code:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

This works. But there is potential problem lurking here. Can you spot it?

Here's one: what if `MAX` is not 5, but 10,000,000? Or 10,000,000,000? Or more? Your memory footprint is pointlessly dreadful: the code here creates a *massive* list, uses it *once*, then throws it away. On top of that, the second `for` loop cannot even *start* until the entire list of squares has been fully calculated. If some poor human is using this program, they'll wonder if the program is stuck.

Even worse: What if you aren't doing arithmetic to get each element - which is fast and cheap - but making a truly expensive calculation? Or making an API call over the network? Or reading from a database? Your program is sluggish, even unresponsive, and might even crash with an out-of-memory error. Its users will think you're a terrible programmer.

The solution is to create an iterator to start with, lazily computing each value only when needed. Then each cycle through the loop happens just in time.

For the record, here is how you create an equivalent iterator class, which fully complies with Python's iterator protocol:

```python
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Holy crap, that's horrible. There's got to be a better way.

Good news: there's a better way. It's called a **generator function**, and you're going to love it!

1.2 Generator Functions

Python provides a tool called the **generator function**, which... well, it's hard to describe everything it gives you in one sentence. Of its many talents, I'll first focus on how it's a *very* useful shortcut for creating iterators.

A generator function looks a lot like a regular function. But instead of saying `return`, it uses a new and different keyword: `yield`. Here's a simple example:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

Use it in a for loop like this:

```
>>> for num in gen_nums():
...     print(num)
0
1
2
3
```

Let's go through and understand this. When you call `gen_nums()` like a function, it immediately returns a **generator object**:

```
>>> sequence = gen_nums()
>>> type(sequence)
<class 'generator'>
```

The *generator function* is `gen_nums` - what we define and then call. A function is a generator function if and only if it uses "yield" instead of "return". The *generator object* is what that generator function returns when called - `sequence`, in this case. A generator function will *always* return a generator object; it can't return anything else. And this generator object is an iterator, which means you can iterate through it using `next()` or a `for` loop:

```
>>> sequence = gen_nums()
>>> next(sequence)
0
>>> next(sequence)
1
>>> next(sequence)
2
>>> next(sequence)
3
>>> next(sequence)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> # Or in a for loop:
... for num in gen_nums(): print(num)
...
0
1
2
3
```

The flow of code works like this: when `next()` is called the first time, or the `for` loop first starts, the body of `gen_nums` starts executing at the beginning, returning the value to the right of the `yield`.

So far, this is much like a regular function. But the next time `next()` is called - or, equivalently, the next time through the `for` loop - the function doesn't start at the beginning again. It starts on the line *after the yield statement*. Look at the source of `gen_nums()` again:

```
def gen_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
```

`gen_nums` is more general than a function or subroutine. It's actually a *coroutine*. You see, a regular function can have several exit points (otherwise known as `return` statements). But it has only one entry point: each time you call a function, it always starts at the first line of the function body.

A coroutine is like a function, except it has several possible *entry* points. It starts with the first line, like a normal function. But when it "returns", the coroutine isn't exiting, so much as *pausing*. Subsequent calls with `next()` - or equivalently, the next time through the `for` loop - start at that `yield` statement again, right where it left off; the re-entry point is the line after the `yield` statement.

And that's the key: **Each `yield` statement simultaneously defines an exit point, *and* a re-entry point.**

For generator objects, each time a new value is requested, the flow of control picks up on the line after the `yield` statement. In this case, the next line increments the variable `n`, then continues with the while loop.

Notice we do not raise `StopIteration` anywhere in the body of `gen_nums()`. When the function body finally exits - after it exits the while loop, in this case - the generator object automatically raises `StopIteration`.

Again: each `yield` statement simultaneously defines an exit point, *and* a re-entry point. In fact, you can have multiple `yield` statements in a generator:

```
def gen_extra_nums():
    n = 0
    while n < 4:
        yield n
        n += 1
    yield 42 # Second yield
```

Here's the output when you use it:

```
>>> for num in gen_extra_nums():
...     print(num)
0
1
2
3
42
```

The second `yield` is reached after the `while` loop exits. When the function reaches the implicit return at the end, the iteration stops. Reason through the code above, and convince yourself it makes sense.

Let's revisit the earlier example, of cycling through a sequence of squares. This is how we first did it:

```
def fetch_squares(max_root):
    squares = []
    for n in range(max_root):
        squares.append(n**2)
    return squares

MAX = 5
for square in fetch_squares(MAX):
    do_something_with(square)
```

As an exercise, pause here, open up a new Python file, and see if you can write a gen_squares generator function that accomplishes the same thing.

Done? Great. Here's what it looks like:

```
>>> def gen_squares(max_num):
...     for num in range(max_num):
...         yield num ** 2
...
>>> MAX = 5
>>> for square in gen_squares(MAX):
...     print(square)
0
1
4
9
16
```

Now, this gen_squares has a problem in Python 2, but not Python 3. Can you spot it?

Here it is: range returns an iterator in Python 3, but in Python 2 it returns a list. If MAX is huge, that creates a huge list inside, killing scalability. So if you are using Python 2, your gen_squares needs to use xrange instead, which acts just like Python 3's range.

The larger point here affects all versions of Python. Generator functions *potentially* have a small memory footprint, but only if you code intelligently. When writing generator functions, be watchful for hidden bottlenecks.

Now, strictly speaking, we don't *need* generator functions for iteration. We just *want* them, because they make certain patterns of scalability far easier. Now that we're in a position to understand it, let's look at the SquaresIterator class again:

```
# Same code we saw earlier.
class SquaresIterator:
    def __init__(self, max_root_value):
        self.max_root_value = max_root_value
        self.current_root_value = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_root_value >= self.max_root_value:
            raise StopIteration
        square_value = self.current_root_value ** 2
        self.current_root_value += 1
        return square_value

# You can use it like this:
for square in SquaresIterator(5):
    print(square)
```

Each value is obtained by invoking its `__next__` method, until it raises `StopIteration`. This produces the same output; but look at the source for the `SquaresIterator` class, and compare it to the source for the generator above. Which is easier to read? Which is easier to maintain? And when requirements change, which is easier to modify without introducing errors? Most people find the generator solution easier and more natural.

Authors often use the word "generator" by itself, to mean either the generator function, *or* the generator object returned when you call it. Typically the writer thinks it's obvious by the context which they are referring to; sometimes it is, sometimes not. Sometimes the writer is not even clear on the distinction to begin with. But it's important: just as there is a big difference between a function, and the value it returns when you call it, so is there a big difference between the generator function, and the generator object it returns.

In your own thought and speech, I encourage you to only use the phrases "generator function" and "generator object", so you are always clear inside yourself, and in your communication. (Which also helps your teammates be more clear.) The only exception: when you truly mean "generator functions and objects", lumping them together, then it's okay to just say "generators". I'll lead by example in this book.

1.3 Generator Patterns and Scalable Composability

Here's a little generator function:

```
def matching_lines_from_file(path, pattern):
    with open(path) as handle:
        for line in handle:
            if pattern in line:
                yield line.rstrip('\n')
```

`matching_lines_from_file()` demonstrates several important practices for modern Python, and is worth studying. It does simple substring matching on lines of a text file, yielding lines containing that substring.

The first line opens a read-only file object, called `handle`. If you haven't been opening your file objects using `with` statements, start today. The main benefit is that once the `with` block is exited, the file object is automatically closed - even if an exception causes a premature exit. It's similar to:

```
try:
    handle = open(path)
    # read from handle
finally:
    handle.close()
```

(The `try`/`finally` is explained in the exceptions chapter.) Next we have `for line in handle`. This useful idiom, which not many people seem to know about, is a special case for text files. Each iteration through the `for` loop, a new line of text will be read from the underlying text file, and placed in the `line` variable.

Sometimes people foolishly take another approach, which I have to warn you about:

```
# Don't do this!!
for line in handle.readlines():
```

`.readlines()` (plural) reads in the *entire file*, parses it into lines, and returns a list of strings - one string per line. By now, you realize how this destroys the generator function's scalability.

Another approach you will sometimes see, which *is* scalable, is to use the file object's `.readline()` method (singular), which manually returns lines one at a time:

```
# .readline() reads and returns a single line of text,
# or returns the empty string at end-of-file.
line = handle.readline()
while line != '':
    # do something with line
    # ...
    # At the end of the loop, read the next line.
    line = handle.readline()
```

But simply writing `for line in handle` is clearer and easier.

After that, it's straightforward: matching lines have any trailing \n-character stripped, and are yielded to the consumer. When writing generator functions, you want to ask yourself "what is the maximum memory footprint of this function, and how can I minimize it?" You can think of scalability as inversely proportional to this footprint. For `matching_lines_from_file()`, it will be about equal to the size of the longest line in the text file. So it's appropriate for the typical human-readable text file, whose lines are small.

(It's also possible to point it to, say, a ten-terabyte text file consisting of exactly one line. If you expect something like *that*, you'll need a different approach.)

Now, suppose a log file contains lines like this:

```
WARNING: Disk usage exceeding 85%
DEBUG: User 'tinytim' upgraded to Pro version
INFO: Sent email campaign, completed normally
WARNING: Almost out of beer
```

... and you exercise `matching_lines_from_file()` like so:

```
for line in matching_lines_from_file("log.txt","WARNING:"):
    print(line)
```

That yields these records:

```
WARNING: Disk usage exceeding 85%
WARNING: Almost out of beer
```

Suppose your application needs that data in dict form:

```
{"level": "WARNING", "message": "Disk usage exceeding 85%"}
{"level": "DEBUG", "message":
    "User 'tinytim' upgraded to Pro version"}
```

We want to scalably transform the records from one form to another - from strings (lines of the log file), to Python dicts. So let's make a new generator function to connect them:

```
def parse_log_records(lines):
    for line in lines:
        level, message = line.split(": ", 1)
        yield {"level": level, "message": message}
```

Now we can connect the two:

```
# log_lines is a generator object
log_lines = matching_lines_from_file("log.txt", "WARNING:")
for record in parse_log_records(log_lines):
    # record is a dict
    print(record)
```

Of course, `parse_log_records()` can be used on its own:

```
with open("log.txt") as handle:
    for record in parse_log_records(handle):
        print(record)
```

`matching_lines_from_file()` and `parse_log_records()` are like building blocks. Properly designed, they can be used to build different data processing streams. I call this *scalable composability*. It goes beyond designing composable functions and types. Ask yourself how you can make the components scalable, **and** whatever is assembled out of them scalable too.

Let's discuss a particular design point. Both `matching_lines_from_file()` and `parse_log_records()` produce an iterator. (Or, more specifically, a generator object). But they have a discrepancy on the input side: `parse_log_records()` accepts an iterator, but `matching_lines_from_file()` requires a path to a file to read from. This means `matching_lines_from_file()` combines two functions: read lines of text from a file, then filter those lines based on some criteria.

Combining functions like this is often what you want in realistic code. But when designing components to flexibly compose together, inconsistent interfaces like this can be limiting. Let's break up the services in `matching_lines_from_file()` into two generator functions:

```
def lines_from_file(path):
    with open(path) as handle:
        for line in handle:
            yield line.rstrip('\n')

def matching_lines(lines, pattern):
    for line in lines:
        if pattern in line:
            yield line
```

You can compose these like so:

```
lines = lines_from_file("log.txt")
matching = matching_lines(lines, 'WARNING:')
for line in matching:
    print(line)
```

Or even redefine `matching_lines_from_file()` in terms of them:

```
def matching_lines_from_file(pattern, path):
    lines = lines_from_file(path)
    matching = matching_lines(lines, pattern)
    for line in matching:
        yield line
```

Conceptually, this factored-out `matching_lines` does a *filtering* operation; all lines are read in, and a subset are yielded. `parse_log_records()` is different. One input record (a `str` line) maps to exactly one output record (a `dict`). Mathematically, it's a *mapping* operation. Think of it as a transformer or adapter. `lines_from_file()` is in a third category; instead of taking a stream as input, it takes a completely different parameter. Since it still returns an iterator of records, think of it as a *source*. And any real program will eventually want to do something with that stream, consuming it without producing another iterator; call that a *sink*.

You need all these pieces to make a working program. When designing a chainable set of generator functions like this - or even better, a toolkit for constructing internal data pipelines - ask yourself whether each component is a sink, a source, or whether it does filtering, or mapping; or whether it's some combination of these. Just asking yourself this question leads to a more usable, readable, and maintainable codebase. And if you're making a library which others will use, you're more likely to end up with a toolkit so powerfully flexible, people use it to build programs you never imagined.

I want you to notice something about `parse_log_records()`. As I said, it fits in the "mapping" category. And notice its mapping is one-to-one: one line of text becomes one dictionary. In other words, each record in the input - a `str` - becomes *exactly one* record in the output - a `dict`.

That isn't always the case. Sometimes, your generator function needs to consume several input records to create one output record. Or the opposite: one input record yields several output records.

Here's an example of the latter. Imagine a text file containing lines in a poem:[1]

```
all night our room was outer-walled with rain
drops fell and flattened on the tin roof
and rang like little disks of metal
```

Let's create a generator function, `words_in_text()`, producing the words one at a time. First approach:

```
# lines is an iterator of text file lines,
# e.g. returned by lines_from_file()
def words_in_text(lines):
    for line in lines:
        for word in line.split():
            yield word
```

This generator function takes a *fan out* approach. No input records are dropped, which means it doesn't do any filtering; it's still purely in the "mapping" category of generator functions. But the mapping isn't one to one. Rather, one input record produces one or more output records. So, when you run the following code:

```
poem_lines = lines_from_file("poem.txt")
poem_words = words_in_text(poem_lines)
for word in poem_words:
    print(word)
```

... it produces this output:

[1]From "Summer Rain", by Amy Lowell. https://www.poets.org/poetsorg/poem/summer-rain

```
all
night
our
room
was
outer-walled
...
```

That first input record - "all night our room was outer-walled with rain" - yields eight words (output records). Ignoring any blank lines in the poem, every line of prose will produce at least one - probably several - words.

The idea of fanning out is interesting, but simple enough. It's more complex when we go the opposite direction: fanning *in*. That means the generator function consumes more than one input record to produce each output record. Doing this successfully requires an awareness of the input's structure, and you'll typically need to encode some simple parsing logic.

Imagine a text file containing residential house sale data. Each record is a set of key-value pairs, one pair per line, with records separated by blank lines:

```
address: 1423 99th Ave
square_feet: 1705
price_usd: 340210

address: 24257 Pueblo Dr
square_feet: 2305
price_usd: 170210

address: 127 Cochran
square_feet: 2068
price_usd: 320500
```

To read this data into a form usable in our code, what we want is a generator function - let's name it `house_records()` - which accepts a sequence of strings (lines) and parses them into convenient dictionaries:

```
>>> lines_of_house_data = lines_from_file("housedata.txt")
>>> houses = house_records(lines_of_house_data)
>>> # Fetch the first record.
... house = next(houses)
>>> house['address']
'1423 99th Ave'
>>> house = next(houses)
>>> house['address']
'24257 Pueblo Dr'
```

How would you create this? If practical, pause here, open up a code editor, and see if you can implement it.

Okay, time's up. Here is one approach:

```
def house_records(lines):
    record = {}
    for line in lines:
        if line == '':
            yield record
            record = {}
            continue
        key, value = line.split(': ', 1)
        record[key] = value
    yield record
```

Notice where the `yield` keywords appear. The last line of the `for` loop reads individual key-value pairs. Starting with an empty record dictionary, it's populated with data until `lines` produces an empty line. That signals the current record is complete, so it's `yield`-ed, and a new record dictionary created. The end of the very last record in `housedata.txt` is signaled not by an empty line, but by the end of the file; that's why we need the final `yield` statement.

As defined, `house_records()` is a bit clunky if we're normally reading from a text file. It makes sense to define a new generator function accepting just the path to the file:

```python
def house_records_from_file(path):
    lines = lines_from_file(path)
    for house in house_records(lines):
        yield house

# Then in your program:
for house in house_records_from_file("housedata.txt"):
    print(house["address"])
```

You may have noticed many of these examples have a bit of boilerplate, when one generator function internally calls another. The last two lines of `house_records_from_file` say:

```python
    for house in house_records(lines):
        yield house
```

Python 3 provides a shortcut, which lets you accomplish this in one line, with the `yield from` statement:

```python
def house_records_from_file(path):
    lines = lines_from_file(path)
    yield from house_records(lines)
```

Even though "yield from" is two words, semantically it's like a single keyword, and distinct from `yield`. The `yield from` statement is used specifically in generator functions, when they yield values directly from another generator object (or, equivalently, by calling another generator function). Using it often simplifies your code, as you see in `house_records_from_file()`. Going back a bit, here's how it works with `matching_lines_from_file()`:

```python
def matching_lines_from_file(pattern, path):
    lines = lines_from_file("log.txt")
    yield from matching_lines(lines, pattern)
```

The formal name for what `yield from` does is "delegating to a sub-generator", and instills a deeper connection between the containing and inner generator objects. In particular, generator objects have certain methods - `send`, `throw` and `close` - for passing information back *into* the context of the running generator function. I won't cover them in this edition of the book, as they are not currently widely used; you can learn more by reading PEPs 342 and 380.[2] If you do use them, `yield from` becomes necessary to enable the flow of information back into the scope of the running coroutine.

[2] https://www.python.org/dev/peps/pep-0342/ and https://www.python.org/dev/peps/pep-0380/

1.4 Python is Filled With Iterators

Let's look at Python 3 dictionaries:[3]

```
>>> calories = {
...     "apple": 95,
...     "slice of bacon": 43,
...     "cheddar cheese": 113,
...     "ice cream": 15, # You wish!
... }
>>> items = calories.items()
>>> type(items)
<class 'dict_items'>
```

So what is this `dict_items` object returned by `calories.items()`? It turns out to be what Python calls a *view*. There is not any kind of base view type; rather, an object quacks like a dictionary view if it supports three things:

- `len(view)` returns the number of items,
- `iter(view)` returns an iterator over the key-value pairs, and
- `(key, value) in view` returns True if that `key-value` pair is in the dictionary, else False.

In other words, a dictionary view is iterable, with a couple of bonus features. It also dynamically updates if its source dictionary changes:

```
>>> items = calories.items()
>>> len(items)
4
>>> calories['orange'] = 50
>>> len(items)
5
>>> ('orange', 50) in items
True
>>> ('orange', 20) in items
False
```

[3] If you're more interested in Python 2, follow along. Every concept in this section fully applies to Python 2.7, with syntax differences we'll discuss at the end.

Dictionaries also have `.keys()` and `.values()`. Like `.items()`, they each return a view. But instead of key-value pairs, they only contain keys or values, respectively:

```
>>> foods = calories.keys()
>>> counts = calories.values()
>>> 'yogurt' in foods
False
>>> 100 in counts
False
>>> calories['yogurt'] = 100
>>> 'yogurt' in foods
True
>>> 100 in counts
True
```

In Python 2 (explained more below), `items()` returns a list of key-value tuples, rather than a view; Python 2 also has an `iteritems()` method that returns an iterator (rather than an iterable view object). Python 3's version of the `items()` method essentially obsoletes both of these. When you do need a list of key-value pairs in Python 3, just write `list(calories.items())`.

Iteration has snuck into many places in Python. The built-in `range` function returns an iterable:

```
>>> seq = range(3)
>>> type(seq)
<class 'range'>
>>> for n in seq: print(n)
0
1
2
```

The built-in `map`, `filter`, and `zip` functions all return iterators:

```
>>> numbers = [1, 2, 3]
>>> big_numbers = [100, 200, 300]
>>> def double(n):
...     return 2 * n
>>> def is_even(n):
...     return n % 2 == 0
>>> mapped = map(double, numbers)
>>> mapped
<map object at 0x1013ac518>
>>> for num in mapped: print(num)
2
4
6
```

```
>>> filtered = filter(is_even, numbers)
>>> filtered
<filter object at 0x1013ac668>
>>> for num in filtered: print(num)
2
```

```
>>> zipped = zip(numbers, big_numbers)
>>> zipped
<zip object at 0x1013a9608>
>>> for pair in zipped: print(pair)
(1, 100)
(2, 200)
(3, 300)
```

Notice that `mapped` is something called a "map object", rather than a list of the results of the calculation; and similar for `filtered` and `zipped`. These are all iterators - giving you all the benefits of iteration, built into the language.

1.4.1 Python 2's Differences

For Python 2, I'll start with some recommendations, before explaining them:

- With dictionaries, always use `viewitems()` rather than `items()` or `iteritems()`. The only exception: if you truly need a list of tuples, use `items()`.

- Likewise for `viewkeys()` and `viewvalues()`, rather than `keys()`, `iterkeys()`, `values()`, and `itervalues()`.
- Use `xrange()` instead of `range()`, unless you have a special need for an actual list.
- Be aware that `map`, `filter`, and `zip` create lists; if your data may grow large, use `imap`, `ifilter` or `izip` from the `itertools` module instead.

These differently named methods and functions essentially have the same behavior as Python 3's more scalable versions. Python 2's `xrange` is just like Python 3's `range`; Python 2's `itertools.imap` is just like Python 3's `map`; and so on.

Let's examine Python 2's dictionary methods. In Python 2, `calories.items()` returns a list of `(key, value)` tuples. So if the dictionary has 10,000 keys, you'd get a list of 10,000 tuples. Similarly, `calories.keys()` returns a list of keys; `calories.values()` returns a list of values. The problems with this will be obvious to you by now: the loop blocks until you create and populate a list, which is immediately thrown away once the loop exits.

Python 2 addressed this by introducing two other methods: `iteritems()`, returning an iterator over the key-value tuples; and (later) `viewitems()`, which returned a view - an iterable type. Similarly, `keys()` gave a list of keys, and they added `iterkeys()` and then `viewkeys()`; and again for `values()`, `itervalues()`, and `viewvalues()`.

In Python 3, what used to be called `viewitems()` was renamed `items()`, and the old `items()` and `iteritems()` went away. Similarly, `keys()` and `values()` were changed to return views instead of lists.

For your own Python 2 code, I recommend you start using `viewitems()`, except when you have an explicit reason to do otherwise. Using `iteritems()` is certainly better than using `items()`, and for Python 2 code, generally works just as well as `viewitems()`. However, if you ever decide to upgrade that codebase with `2to3`, the resulting code will be closer to your original program.[4] Python 2's `viewitems` basically obsoleted `iteritems`, which is why the latter has no equivalent in Python 3.

The situation with `range` is simpler. Python 2's original `range()` function returned a list; later, `xrange()` was added, returning an iterable, and practically speaking obsoleting Python 2's `range()`. But many people continue to use `range()`, for a range (ha!) of reasons. Python 3's version of `range()` is essentially the same as Python 2's `xrange()`, and Python 3 has no

[4] `2to3` will replace both `iteritems()` and `viewitems()` with `items()`; but the precise semantics of the converted program will more closely match your Python 2 code if you use `viewitems()` to begin with.

function named xrange. (Of course, list(range(. . .)) will give you an actual list, if you need it.)

map, filter, and zip are well used in certain circles. If you want your Python 2 code using these functions to be fully forward-compatible, you have to go to a little more trouble: their iterator-equivalents are all in the itertools module. So, instead of this:

```
mapped = map(double, numbers)
```

you will need to write this:

```
from itertools import imap
mapped = imap(double, numbers)
```

The 2to3 program will convert Python 2's imap(f, items) to map(f, items), but will convert Python 2's map(f, items) to list(map(f, items)). The itertools module similarly has ifilter and izip, for which the same patterns apply.

It's important to realize that everything described for Python 3 also applies to Python 2.7, *if* you use the different names of the relevant methods and functions. And that is what I recommend you do, so you get the scalability benefits of iterators, and have an easier transition to Python 3.

1.5 The Iterator Protocol

This optional section explains Python's **iterator protocol** in formal detail, giving you a precise and low-level understanding of how generators, iterators, and iterables all work. For the day-to-day coding of most programmers, it's not nearly as important as everything else in this chapter. That said, you need this information to implement your own, custom iterable collection types. Personally, I also find knowing the protocol helps me reason through iteration-related issues and edge cases; by knowing these details, I'm able to quickly troubleshoot and fix certain bugs that might otherwise eat up my afternoon. If this all sounds valuable to you, keep reading; otherwise, feel free to skip to the next chapter.

As mentioned, Python makes a distinction between *iterators*, versus objects that are *iterable*. The difference is subtle to begin with, and frankly it doesn't help that the two words sound nearly identical. Keep clear in your mind that "iterator" and "iterable" are distinct but related concepts, and the following will be easier to understand.

Informally, an iterator is something you can pass to `next()`, or use exactly once in a `for` loop. More formally, an object in Python 3 is an iterator if follows the **iterator protocol**. And an object follows the iterator protocol if it meets the following criteria:

- It defines a method named `__next__`, called with no arguments.
- Each time `__next__()` is called, it produces the next item in the sequence.
- Until all items have been produced. Then, subsequent calls to `__next__()` raise `StopIteration`.
- It also defines a boilerplate method named `__iter__`, called with no arguments, and returning the same iterator. Its body is literally `return self`.

Any object with these methods can call itself a Python iterator. You are not intended to call the `__next__()` method directly. Instead, you will use the built-in `next()` function. To understand better, here is a simplified way you might write your own `next()` function:

```
def next(it, default=None):
    try:
        return it.__next__()
    except StopIteration:
        if default is None:
            raise
        return default
```

(This version will not let you specify `None` as a default value; the real built-in `next()` will. Otherwise, it's essentially accurate.)

All the above has one difference in Python 2: the iterator's method is named `.next()` rather than `.__next__()`. Abstracting over this difference is one reason to use the built-in, top-level `next()` function. Of course, using `next()` lets you specify a default value, whereas `.__next__()` and `.next()` do not.

Now, all the above is for the "iterator". Let's explain the other word, "iterable". Informally, an object is *iterable* if it knows how to create an iterator over its contents, which you can access with the built-in `iter()` function. More formally, a Python container object is *iterable* if it meets one of these two criteria:

- It defines a method called `__iter__()`, which creates and returns an iterator over the elements in the container; or

- it follows the *sequence protocol*. This means it defines `__getitem__` - the magic method for square brackets - and lets you reference `foo[0]`, `foo[1]`, etc., raising an `IndexError` once you go past the last element.

(Notice "iterator" is a noun, while "iterable" is usually an adjective. This can help you remember which is which.)

When implementing your own container type, you probably want to make it iterable, so you and others can use it in a `for` loop. Depending on the nature of the container, it's often easiest to implement the sequence protocol. As an example, consider this simple `UniqueList` type, which is a kind of hybrid between a list and a set:

```
class UniqueList:
    def __init__(self, items):
        self.items = []
        for item in items:
            self.append(item)
    def append(self, item):
        if item not in self.items:
            self.items.append(item)
    def __getitem__(self, index):
        return self.items[index]
```

Use it like this:

```
>>> u = UniqueList([3,7,2,9,3,4,2])
>>> u.items
[3, 7, 2, 9, 4]
>>> u[3]
9
>>> u[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in __getitem__
IndexError: list index out of range
```

The `__getitem__` method implements square-bracket access; basically, Python translates `u[3]` into `u.__getitem__(3)`. We've programmed this object's square brackets to operate much like a normal list, in that the initial element is at index 0, and you get subsequent elements with subsequent integers, not skipping any. And when you go past the end, it raises `IndexError`. If an object has a `__getitem__` method behaving in just this way, `iter()` knows how to create an iterator over it:

```
>>> u_iter = iter(u)
>>> type(u_iter)
<class 'iterator'>
>>> for num in u_iter: print(num)
3
7
2
9
4
```

Notice we get a lot of this behavior for free, simply because we're using an actual list internally (and thus delegating much of the `__getitem__` logic to it). That's a clue for you, whenever you make a custom collection that acts like a list - or one of the other standard collection types. If your object internally stores its data in one of the standard data types, you'll often have an easier time mimicking its behavior.

Implementing the sequence protocol - i.e., writing a `__getitem__` method which acts like a list's - is one way to make your class iterable. The other involves writing an `__iter__` method. When called with no arguments, it must return some object which follows the iterator protocol, described above. In the worst case, you'll need to implement something like the `SquaresIterator` class from earlier in this chapter, with its own `__next__` and `__iter__` methods. But usually you don't have to work that hard, because you can simply return a generator object instead. That means `__iter__` is a generator function itself, or it internally calls some other generator function, returning its value.

Both iterables and iterators must have an `__iter__` method. Both are called with no argument, and both return an iterator object. The only difference: the one for the iterator returns its `self`, while an iterable's will create and return a *new* iterator. And if you call it twice, you get two different iterators.

This similarity is intentional, to simplify control code that can accept either iterators or iterables. Here's the mental model you can safely follow: when Python's runtime encounters a `for` loop, it will start by invoking `iter(sequence)`. This *always* returns an iterator: either `sequence` itself, or (if `sequence` is only iterable) the iterator created by `sequence.__iter__()`.

Iterables are everywhere in Python. Almost all built-in collection types are iterable: `list`, `tuple`, and `set`, and even `dict` (though more often you'll want to use `dict.items()` or dict.viewitems()). In your own custom collection classes, sometimes the easiest way to implement `__iter__()` actually involves using `iter()`. For instance, this will not work:

```
class BrokenInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        return self.items
```

If you try it, you get a `TypeError`:

```
>>> items = BrokenInLoops()
>>> for item in items:
...     print(item)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: iter() returned non-iterator of type 'list'
```

It doesn't work because `__iter__()` is supposed to return an iterator, but a list object is *not* an iterator; it is simply *iterable*. You can fix this with a one-line change: use `iter()` to create an iterator object inside of `__iter__()`, and return that object:

```
class WorksInLoops:
    def __init__(self):
        self.items = [7, 3, 9]
    def __iter__(self):
        # This class is identical to BrokenInLoops,
        # except for this next line.
        return iter(self.items)
```

This makes `WorksInLoops` itself iterable, because `__iter__` returns an actual iterator object - making `WorksInLoops` follow the iterator protocol correctly. That `__iter__` method generates a fresh iterator each time:

```
>>> items = WorksInLoops()
>>> for item in items:
...     print(item)
7
3
9
```

Chapter 2

Creating Collections with Comprehensions

A *list comprehension* is a high level, declarative way to create a list in Python. They look like this:

```
>>> squares = [ n*n for n in range(6) ]
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

This is exactly equivalent to the following:

```
>>> squares = []
>>> for n in range(6):
...     squares.append(n*n)
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice that in the first example, what you type is declaring *what* kind of list you want, while the second is specifying *how* to create it. That's why we say it is high-level and declarative: it's as if you are stating what kind of list you want created, and then let Python figure out how to build it.

Python lets you write other kinds of comprehensions other than lists. Here's a simple dictionary comprehension, for example:

```
>>> blocks = { n: "x" * n for n in range(5) }
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

This is exactly equivalent to the following:

```
>>> blocks = dict()
>>> for n in range(5):
...     blocks[n] = "x" * n
>>> print(blocks)
{0: '', 1: 'x', 2: 'xx', 3: 'xxx', 4: 'xxxx'}
```

The main benefits of comprehensions are readability and maintainability. Most people find them *very* readable; even developers encountering a comprehension for the first time will usually find their first guess about what it means to be correct. You can't get more readable than that.

And there is a deeper, cognitive benefit: once you've practiced with them a bit, you will find you can write them with very little mental effort - keeping more of your attention free for other tasks.

Beyond lists and dictionaries, there are several other forms of comprehension you will learn about in this chapter. As you become comfortable with them, you will find them to be versatile and very Pythonic - meaning, you'll find they fit well into many other Python idioms and constructs, lending new expressiveness and elegance to your code.

2.1 List Comprehensions

A list comprehension is the most widely used and useful kind of comprehension, and is essentially a way to create and populate a list. Its structure looks like:

```
[ EXPRESSION for VARIABLE in SEQUENCE ]
```

EXPRESSION is any Python expression, though in useful comprehensions, the expression typically has some variable in it. That variable is stated in the *VARIABLE* field. *SEQUENCE* defines the source values the variable enumerates through, creating the final sequence of calculated values.

Here's the simple example we glimpsed earlier:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
>>> print(squares)
[0, 1, 4, 9, 16, 25]
```

Notice the result is just a regular list. In `squares`, the expression is `n*n`; the variable is `n`; and the source sequence is `range(6)`. The sequence is a `range` object; in fact, it can be any iterable… another list or tuple, a generator object, or something else.

The expression part can be anything that reduces to a value:

- Arithmetic expressions like `n+3`
- A function call like `f(m)`, using `m` as the variable
- A slice operation (like `s[::-1]`, to reverse a string)
- Method calls (`foo.bar()`, iterating over a sequence of objects)
- And more.

Some complete examples:

```
>>> # First define some source sequences...
... pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> # And a helper function...
... def repeat(s):
...     return s + s
...
>>> # Now, some list comprehensions:
... [ 2*m+3 for m in range(10, 20, 2) ]
[23, 27, 31, 35, 39]
>>> [ abs(num) for num in numbers ]
[9, 1, 4, 20, 11, 3]
>>> [ 10 - x for x in numbers ]
[1, 11, 14, -10, -1, 13]
>>> [ pet.lower() for pet in pets ]
['dog', 'parakeet', 'cat', 'llama']
>>> [ "The " + pet for pet in sorted(pets) ]
['The cat', 'The dog', 'The llama', 'The parakeet']
>>> [ repeat(pet) for pet in pets ]
['dogdog', 'parakeetparakeet', 'catcat', 'llamallama']
```

Notice how all these fit the same structure. They all have the keywords "for" and "in"; those are required in Python, for any kind of comprehension you may write. These are interleaved among three fields: the expression; the variable (i.e., the identifier from which the expression is composed); and the source sequence.

The order of elements in the final list is determined by the order of the source sequence. But you can filter out elements by adding an "if" clause:

```
>>> def is_palindrome(s):
...     return s == s[::-1]
...
>>> pets = ["dog", "parakeet", "cat", "llama"]
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> words = ["bib", "bias", "dad", "eye", "deed", "tooth"]
>>>
>>> [ n*2 for n in numbers if n % 2 == 0 ]
[-8, 40]
>>>
>>> [pet.upper() for pet in pets if len(pet) == 3]
['DOG', 'CAT']
>>>
>>> [n for n in numbers if n > 0]
[9, 20, 11]
>>>
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

The structure is

```
[ EXPR for VAR in SEQUENCE if CONDITION ]
```

where *CONDITION* is an expression that evaluates to `True` or `False`, depending on the variable.[1] Note that it can be either a function applied to the variable (`is_palindrome(word)`), or a more complex expression. Choosing to use a function can improve readability, and also let you apply filter logic whose code won't fit on one line.

A list comprehension must always have the "for" word, even if the beginning expression is just the variable itself. For example, when we say:

```
>>> [word for word in words if is_palindrome(word)]
['bib', 'dad', 'eye', 'deed']
```

Sometimes people think `word for word in words` seems redundant (it does), and try to shorten it... but that doesn't work:

[1]Technically, the condition doesn't have to depend on the variable, but it's hard to imagine building a useful list comprehension this way.

```
>>> [word in words if is_palindrome(word)]
  File "<stdin>", line 1
    [word in words if is_palindrome(word)]
                                         ^
SyntaxError: invalid syntax
>>>
```

2.2 Formatting For Readability (And More)

Realistic list comprehensions tend to be too long to fit nicely on a single line. And they are composed of distinct logical parts, which can vary independently as the code evolves. This creates a couple of inconveniences, which are solved by a very convenient fact: *Python's normal rules of whitespace are suspended inside the square brackets*. You can exploit this to make them more readable and maintainable, splitting them across multiple lines:

```
def double_short_words(words):
    return [ word + word
             for word in words
             if len(word) < 5 ]
```

Another variation, which some people prefer:

```
def double_short_words(words):
    return [
        word + word
        for word in words
        if len(word) < 5
        ]
```

What I've done here is split the comprehension across separate lines. You can, and should, do this with any substantial comprehension. It's great for several reasons, the most important being the instant gain in readability. This comprehension has three separate ideas expressed inside the square brackets: the expression (`word + word`); the sequence (`for word in words`); and the filtering clause (`if len(word) < 5`). These are logically separate aspects, and by splitting them across different lines, it takes less cognitive effort for a human to read and understand than the one-line version. It's effectively pre-parsed for you, as you read the code.

There's another benefit: version control and code review diffs are more pin-pointed. Imagine you and I are on the same development team, working on this code base in different feature branches. In my branch, I change the expression to `"word + word + word"`; in yours, you change the threshold to `"len(word) < 7"`. If the comprehension is on one line, version control tools will perceive this as a merge conflict, and whoever merges last will have to manually fix it.[2] But since this list comprehension is split across three lines, our source control tool can automatically merge both our branches. And if we're doing code reviews like we should be, the reviewer can identify the precise change immediately, without having to scan the line and think.

2.3 Multiple Sources and Filters

You can have several `for VAR in SEQUENCE` clauses. This lets you construct lists based on pairs, triplets, etc., from two or more source sequences:

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard',
 'orange doll', 'purple bike', 'purple basketball',
 'purple skateboard', 'purple doll', 'pink bike',
 'pink basketball', 'pink skateboard', 'pink doll']
```

Every pair from the two sources, `colors` and `toys`, is used to calculate a value in the final list. That final list has 12 elements, the product of the lengths of the two source lists.

I want you to notice that the two `for` clauses are independent of each other; `colors` and `toys` are two unrelated lists. Using multiple `for` clauses can sometimes take a different form, where they are more interdependent. Consider this example:

[2] I like to think future version control tools will automatically resolve this kind of situation. I believe it will require the tool to have some knowledge of the language syntax, so it can parse and reason about different clauses in a line of code.

```
>>> ranges = [range(1,7), range(4,12,3), range(-5,9,4)]
>>> [ float(num)
...   for subrange in ranges
...   for num in subrange ]
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 4.0, 7.0, 10.0, -5.0,
-1.0, 3.0, 7.0]
```

The source sequence - `"ranges"` - is a list of range objects.[3] Now, this list comprehension has two `for` clauses again. But notice one depends on the other. The source of the second is the variable for the first!

It's not like the colorful-toys example, whose `for` clauses are independent of each other. When chained together this way, order matters:

```
>>> [ float(num)
...   for num in subrange
...   for subrange in ranges ]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'subrange' is not defined
```

Python parses the list comprehension from left to right. If the first clause is `for num in subrange`, at that point `subrange` is not defined. So you have to put `for subrange in ranges` *first*. You can chain more than two `for` clauses together like this; the first one will just need to reference a previously-defined source, and the others can use sources defined in the previous `for` clause, like `subrange` is defined.

Now, that's for chained `for` clauses. If the clauses are independent, does the order matter at all? It does, just in a different way. What's the difference between these two list comprehensions:

[3] Refresher: The `range` built-in returns an iterator over a sequence of integers, and can be called with 1, 2, or 3 arguments. The most general form is `range(start, stop, step)`, beginning at `start`, going up to *but not including* `stop`, in increments of `step`. Called with 2 arguments, the step-size defaults to one; with one argument, that argument is the `stop`, and the sequence starts at zero. In Python 2, use `xrange` instead of `range`.

```
>>> colors = ["orange", "purple", "pink"]
>>> toys = ["bike", "basketball", "skateboard", "doll"]
>>>
>>> [ color + " " + toy
...   for color in colors
...   for toy in toys ]
['orange bike', 'orange basketball', 'orange skateboard',
'orange doll', 'purple bike', 'purple basketball',
'purple skateboard', 'purple doll', 'pink bike',
'pink basketball', 'pink skateboard', 'pink doll']
>>>
>>> [ color + " " + toy
...   for toy in toys
...   for color in colors ]
['orange bike', 'purple bike', 'pink bike', 'orange
basketball', 'purple basketball', 'pink basketball',
'orange skateboard', 'purple skateboard', 'pink
skateboard', 'orange doll', 'purple doll', 'pink doll']
```

The order here doesn't matter in the sense it does for chained `for` clauses, where you *must* put things in a certain order, or your program won't run. Here, you have a choice. And that choice *does* effect the order of elements in the final comprehension. The first element in each is "orange bike". And notice the second element is different. Think a moment, and ask yourself: why? Why is the first element the same in both comprehensions? And why is it only the *second* element that's different?

It has to do with which sequence is held constant while the other varies. It's the same logic that applies when nesting regular `for` loops:

```
>>> # Nested one way...
... build_colors_toys = []
>>> for color in colors:
...     for toy in toys:
...         build_colors_toys.append(color + " " + toy)
>>> build_colors_toys[0]
'orange bike'
>>> build_colors_toys[1]
'orange basketball'
>>>
>>> # And nested the other way.
... build_toys_colors = []
>>> for toy in toys:
...     for color in colors:
...         build_toys_colors.append(color + " " + toy)
>>> build_toys_colors[0]
'orange bike'
>>> build_toys_colors[1]
'purple bike'
```

The second `for` clause in the list comprehension corresponds to the innermost `for` loop. Its values vary through their range more rapidly than the outer one.

In addition to using many `for` clauses, you can have more than one `if` clause, for multiple levels of filtering. Just write several of them in sequence:

```
>>> numbers = [ 9, -1, -4, 20, 17, -3 ]
>>> odd_positives = [
...     num for num in numbers
...     if num > 0
...     if num % 2 == 1
... ]
>>> print(odd_positives)
[9, 17]
```

Here, I've placed each `if` clause on its own line, for readability - but I could have put both on one line. When you have more than one `if` clause, each element must meet the criteria of *all* of them to make it into the final list. In other words, `if` clauses are "and-ed" together, not "or-ed" together.

What if you want to do "or" - to include elements matching at least one of the `if` clause criteria, omitting only those not matching either? List comprehensions don't allow you do to that directly. The comprehension mini-language is not as expressive as Python itself, and there are lists you might need to construct which cannot be expressed as a comprehension.

But sometimes you can cheat a bit by defining helper functions. For example, here's how you can filter based on whether the number is a multiple of 2 **or** 3:

```
>>> numbers = [ 9, -1, -4, 20, 11, -3 ]
>>> def is_mult_of_2_or_3(num):
...     return (num % 2 == 0) or (num % 3 == 0)
...
>>> [
...     num for num in numbers
...     if is_mult_of_2_or_3(num)
... ]
[9, -4, 20, -3]
```

We discuss this more in the "Limitations" section, later in the chapter.

You can use multiple `for` and `if` clauses together:

```
>>> weights = [0.2, 0.5, 0.9]
>>> values = [27.5, 13.4]
>>> offsets = [4.3, 7.1, 9.5]
>>>
>>> [ (weight, value, offset)
...   for weight in weights
...   for value in values
...   for offset in offsets
...   if offset > 5.0
...   if weight * value < offset ]
[(0.2, 27.5, 7.1), (0.2, 27.5, 9.5), (0.2, 13.4, 7.1),
(0.2, 13.4, 9.5), (0.5, 13.4, 7.1), (0.5, 13.4, 9.5)]
```

The only rule is that the first `for` clause must come before the first `if` clause. Other than that, you can interleave `for` and `if` clauses in any order, though most people seem to find it more readable to group all the `for` clauses together at first, then the `if` clauses together at the end.

2.4 Comprehensions and Generators

List comprehensions create lists:

```
>>> squares = [ n*n for n in range(6) ]
>>> type(squares)
<class 'list'>
```

When you need a list, that's great, but sometimes you don't *need* a list, and you'd prefer something more scalable. It's like the situation near the start of the generators chapter:

```
# This again.
NUM_SQUARES = 10*1000*1000
many_squares = [ n*n for n in range(NUM_SQUARES) ]
for number in many_squares:
    do_something_with(number)
```

The entire `many_squares` list must be fully created - all memory for it must be allocated, and every element calculated - before `do_something_with` is called even *once*. And memory usage goes through the roof.

You know one solution: write a generator function, and call it. But there's an easier option: write a *generator expression*. This is the official name for it, but it really should be called a "generator comprehension". Syntactically, it looks just like a list comprehension - except you use parentheses instead of square brackets:

```
>>> generated_squares = ( n*n for n in range(NUM_SQUARES) )
>>> type(generated_squares)
<class 'generator'>
```

This "generator expression" creates a *generator object*, in the exact same way a list comprehension creates a list. Any list comprehension you can write, you can use to create an equivalent generator object, just by swapping `"("` and `")"` for `"["` and `"]"`.

And you're creating the object directly, without having to define a generator function to call. In other words, a generator expression is a convenient shortcut when you need a quick generator object:

```
# This...
many_squares = ( n*n for n in range(NUM_SQUARES) )

# ... is EXACTLY EQUIVALENT to this:
def gen_many_squares(limit):
    for n in range(limit):
        yield n * n
many_squares = gen_many_squares(NUM_SQUARES)
```

As far as Python is concerned, there is no difference.

Everything you know about list comprehensions applies to generator expressions: multiple `for` clauses, `if` clauses, etc. You only need to type the parentheses.

In fact, sometimes you can even omit them. When passing a generator expression as an argument to a function, you will sometimes find yourself typing `"(("` followed by `"))"`. In that situation, Python lets you omit the inner pair. Imagine, for example, you are sorting a list of customer email addresses, looking at only those customers whose status is "active":

```
>>> # User is a class with "email" and "is_active" fields.
... # all_users is a list of User objects.

>>> # Sorted list of active user's email addresses.
... # Passing in a generator expression.
>>> sorted((user.email for user in all_users
...          if user.is_active))
['fred@a.com', 'sandy@f.net', 'tim@d.com']
>>>
>>> # Omitting the inner parentheses.
... # Still passing in a generator expression!
>>> sorted(user.email for user in all_users
...        if user.is_active)
['fred@a.com', 'sandy@f.net', 'tim@d.com']
```

Notice how readable and natural this is (or will be, once you've practiced a bit). One thing to watch out for: you can only inline a generator expression this way when passed to a function or method of one argument. Otherwise, you get a syntax error:

```
>>>
>>> # Reverse that list. Whoops...
... sorted(user.email for user in all_users
...         if user.is_active, reverse=True)
  File "<stdin>", line 2
SyntaxError: Generator expression must be parenthesized if not sole
   argument
```

Python can't unambiguously interpret what you mean here, so you must use the inner parentheses:

```
>>> # Okay, THIS will get the reversed list.
... sorted((user.email for user in all_users
...         if user.is_active), reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

And of course, sometimes it's more readable to assign the generator expression to a variable:

```
>>> active_emails = (
...        user.email for user in all_users
...        if user.is_active
... )

>>> sorted(active_emails, reverse=True)
['tim@d.com', 'sandy@f.net', 'fred@a.com']
```

Generator expressions without parentheses suggest a unified way of thinking about comprehensions, which link generator expressions and list comprehensions together. Here's a generator expression for a sequence of squares:

```
( n**2 for n in range(10) )
```

And here it is again, passed to the built-in `list()` function:

```
list( n**2 for n in range(10) )
```

And here it is as a list comprehension:

```
[ n**2 for n in range(10) ]
```

When you understand generator expressions, it's easy to see list comprehensions as a derivative data structure. And the same applies for dictionary and set comprehensions (covered next). With this insight, you start seeing new opportunities to use all of them in your own code, improving its readability, maintainability, and performance in the process.

2.4.1 Generator Expression or List Comprehension?

If generator expressions are so great, why would you use list comprehensions? Generally speaking, when deciding which to use, your code will be more scalable and responsive if you use a generator expression. Except, of course, when you actually need a list. There are several considerations.

First, if the sequence is unlikely to be very big - and by big, I mean a minimum of thousands of elements long - you probably won't benefit from using a generator expression. That's just not big enough for scalability to matter. They're also immutable. If you need random access, or to go through the sequence twice, or you might need to append or remove elements, generator expressions won't work.

This is especially important when writing methods or functions whose return value is a sequence. Do you return a generator expression, or a list comprehension? In theory, there's no reason to ever return a list instead of a generator object; a list can be trivially created by passing it to `list()`. In practice, the interface may be such that the caller will really want an actual list. Also, if you are constructing the return value as a list within the function, it's silly to return a generator expression over it - just return the actual list.

And if your intention is to create a library usable by people who may not be advanced Pythonistas, that can be an argument for returning lists. Almost all programmers are familiar with list-like data structures. But fewer are familiar with how generators work in Python, and may - quite reasonably - get confused when confronted with a generator object.

2.5 Dictionaries, Sets, and Tuples

Just like a list comprehension creates a list, a dictionary comprehension creates a dictionary. You saw an example at the beginning of this chapter; here's another. Suppose you have this `Student` class:

```
class Student:
    def __init__(self, name, gpa, major):
        self.name = name
        self.gpa = gpa
        self.major = major
```

Given a list `students` of student objects, we can write a dictionary comprehension mapping student names to their GPAs:

```
>>> { student.name: student.gpa for student in students }
{'Jim Smith': 3.6, 'Ryan Spencer': 3.1,
 'Penny Gilmore': 3.9, 'Alisha Jones': 2.5,
 'Todd Reynolds': 3.4}
```

The syntax differs from that of list comprehensions in two ways. Instead of square brackets, you're using curly brackets - which makes sense, since this creates a dictionary. The other difference is the expression field, whose format is "key: value", since a dict has key-value pairs. So the structure is

```
{ KEY : VALUE for VARIABLE in SEQUENCE }
```

These are the only differences. *Everything* else you learned about list comprehensions applies, including filtering with if clauses:

```
>>> def invert_name(name):
...     first, last = name.split(" ", 1)
...     return last + ", " + first
...
>>> # Get "lastname, firstname" of high-GPA students.
... { invert_name(student.name): student.gpa
...   for student in students
...   if student.gpa > 3.5 }
{'Smith, Jim': 3.6, 'Gilmore, Penny': 3.9}
```

You can create sets too. Set comprehensions look exactly like a list comprehension, but with curly braces instead of square brackets:

```
>>> # A list of student majors...
... [ student.major for student in students ]
['Computer Science', 'Economics', 'Computer Science',
 'Economics', 'Basket Weaving']
>>> # And the same as a set:
... { student.major for student in students }
{'Economics', 'Computer Science', 'Basket Weaving'}
>>> # You can also use the set() built-in.
... set(student.major for student in students)
{'Economics', 'Computer Science', 'Basket Weaving'}
```

(How does Python distinguish between a set and dict comprehension? Because the dict's expression is a key-value pair, while a set's has a single value.)

What about tuple comprehensions? This is fun: strictly speaking, Python doesn't support them. However, you can pretend it does by using tuple():

```
>>> tuple(student.gpa for student in students
...       if student.major == "Computer Science")
(3.6, 2.5)
```

This creates a tuple, *but it's not a tuple comprehension*. You're calling the `tuple` constructor, and passing it a single argument. What's that argument? A generator expression! In other words, you're doing this:

```
>>> cs_students = (
...     student.gpa for student in students
...     if student.major == "Computer Science"
...     )
>>> type(cs_students)
<class 'generator'>
>>> tuple(cs_students)
(3.6, 2.5)
>>>
>>> # Same as:
... tuple((student.gpa for student in students
...        if student.major == "Computer Science"))
(3.6, 2.5)
>>> # But you can omit the inner parentheses.
```

`tuple`'s constructor takes an iterator as an argument. The `cs_students` is a generator object (created by the generator expression), and a generator object is an iterator. So you can *pretend* like Python has tuple comprehensions, using `"tuple("` as the opener and `")"` as the close. In fact, this also gives you alternate ways to create dictionary and set comprehensions:

```
>>> # Same as:
... # { student.name: student.gpa for student in students }
>>> dict((student.name, student.gpa)
...      for student in students)
{'Jim Smith': 3.6, 'Penny Gilmore': 3.9,
 'Alisha Jones': 2.5, 'Ryan Spencer': 3.1,
 'Todd Reynolds': 3.4}
>>> # Same as:
... # { student.major for student in students }
>>> set(student.major for student in students)
{'Computer Science', 'Basket Weaving', 'Economics'}
```

Remember, when you pass a generator expression into a function, you can omit the inner parentheses. That's why you can, for example, type

```
tuple(f(x) for x in numbers)
```

instead of

```
tuple((f(x) for x in numbers))
```

One last point. Generator expressions are a scalable analogue of list comprehensions; is there any such equivalent for dicts or sets? No, it turns out. If you need to lazily generate key-value pairs or unique elements, your best bet is to write a generator function.

2.6 Limits of Comprehensions

Comprehensions have a few wrinkles people sometimes trip over. Consider the following code:

```
# Read in the lines of a file, stripping leading and
# trailing whitespace, skipping any empty or
# whitespace-only lines.
trimmed_lines = []
for line in open('wombat-story.txt'):
    line = line.strip()
    if line != "":
        trimmed_lines.append(line)

print("Got {} lines".format(len(trimmed_lines)))
```

Straightforward enough - we're building a list named `trimmed_lines`. The resulting list has all leading and trailing whitespace removed from its elements, skipping any empty lines (or lines that were just whitespace to begin with). It's not hard to imagine needing to do something like this in a real program.

Now… how would you do this using a list comprehension? Here's a first try:

```
with open('wombat-story.txt') as story:
    trimmed_lines = [
        line.strip()
        for line in story
        if line.strip() != ""
        ]

print("Got {} lines".format(len(trimmed_lines)))
```

This works. Notice, though, that `line.strip()` appears twice. That's wasting CPU cycles compared to the for-loop version, which only calls `line.strip()` once. Stripping whitespace from a string isn't *that* expensive, computationally speaking. But sooner or later, you will want to do something where this matters:

```
>>> values = [
...     expensive_function(n)
...     for n in range(BIG_NUMBER)
...     if expensive_function(n) > 0
...     ]
```

So how can you create this as a list comprehension, while calling `expensive_function` only once? It turns out there is no direct way to do this. There some perhaps-too-clever solutions, such as memoizing (which can easily overuse memory), nesting a generator expression inside (probably the best choice), or making an intermediate list (if it's small).

If the sequence you need fits the pattern above, you might consider building it the old-fashioned way, using a `for` loop or a generator function. Or if you still want to use a comprehension, use an intermediate generator expression. The result is fairly readable and understandable (at least for you, having read this far):

```
>>> intermediate_values = (
...     expensive_function(n)
...     for n in range(10000)
... )
>>>
>>> values = [
...     intermediate_value
...     for intermediate_value in intermediate_values
...     if intermediate_value > 0
...     ]
```

Another limitation is that comprehensions must be built on one element at a time. The best way to see this is to imagine a list composed of inlined key-value pairs - flattened, in other words, so even-indexed elements are keys, and each key's value comes right after it. Imagine a function that converts this to a dictionary:

```
>>> # Price per pound of fruits & vegetables, in dollars.
... prices_flat_list = [
...     "orange", 0.70, "banana", 0.86,
...     "cantaloupe", 0.63, "bok choy", 1.56,
...     "coconuts", 1.06 ]

>>> list2dict(prices_flat_list)
{'banana': 0.86, 'bok choy': 1.56, 'cantaloupe': 0.63, 'orange':
   0.7, 'coconuts': 1.06}
```

Here's one way to implement list2dict:

```
# Converts a "flattened" list into an "unflattened" dict.
def list2dict(flattened):
    assert len(flattened) % 2 == 0,
        "Input must be list of key-value pairs"
    unflattened = dict()
    for i in range(0, len(flattened), 2):
        key, value = flattened[i], flattened[i+1]
        unflattened[key] = value
    return unflattened
```

Look at list2dict's for loop. It runs through the even index numbers of elements in flattened, rather than the elements of flattened directly. This allows it to refer to two different list elements each time through the for loop. But this turns out to be something which just can't be expressed in the semantics of a Python comprehension. Generally, a comprehension operates by looking at each element in some source sequence, one at a time; it can't peek at neighboring elements.

Another example: a function grouping the elements of a sequence by some criteria - for example, the first letter of a string:

```
>>> names = ["Joe", "Jim", "Todd",
...     "Tiffany", "Zelma", "Gerry", "Gina"]
>>> grouped_names = group_by_first_letter(names)
>>> grouped_names['j']
['Joe', 'Jim']
>>> grouped_names['z']
['Zelma']
```

Here's one way to implement the grouping function:

```
from collections import defaultdict
def group_by_first_letter(items):
    grouped = defaultdict(list)
    for item in items:
        key = item[0].lower()
        grouped[key].append(item)
    return grouped
```

Again, the semantics of Python comprehensions aren't built to support this kind of algorithm. In functional programming terms, comprehensions can use map and filter operations, but not reduce or fold. Fortunately, this covers many use cases. I point out these limitations to help you avoid wasting time trying to figure them out; in spite of them, I find comprehensions to be a valuable part of my daily Python toolbox, and I'm sure you will too.

Index