Chapter 20. Comprehensions and Generations

This chapter explores a set of advanced function-related tools and topics. Its main subjects are *generator functions* and their *generator expression* relatives—user-defined ways to produce results on demand the same way that many built-ins do. Because generators apply the *iteration* protocol and generator expressions reuse *comprehension* syntax, this chapter is also partly a follow-up to Chapter 14 (hence its title). We'll extend these topics to their completion here and demo with larger examples that tie ideas together.

Finally, this chapter provides just enough of an intro to get you started with `async` *coroutines*—tools that build on generators, but assume knowledge of parallel programming, which is outside the scope of this book and the needs of most Python learners. You won't become an `async` master here, but you'll get a head start for further explorations.

Iteration and generation in Python also encompasses user-defined *classes*, but we'll defer that final part of this story until Part VI, when we study operator overloading. The next chapter continues threads spun here by timing the relative performance of some of this chapter's tools as a larger case study. Before that, though, let's pick up the comprehensions and iterations story where it was last left, and extend it to include value generators.

Comprehensions: The Final Act

As mentioned early in this book, Python supports the procedural, object-oriented, and function programming paradigms. Among these, Python has a host of tools that most observers would consider *functional* in nature, which we enumerated in the preceding chapter—closures, generators, lambdas, comprehensions, maps, decorators, function objects, and more. These tools allow us to apply and combine functions in powerful ways, and often offer state retention and coding solutions that are alternatives to classes and OOP.

For instance, the prior chapter explored tools such as `map` and `filter` — key members of Python's early functional-programming toolset inspired by the Lisp language—that map operations over iterables and collect results. Because this is such a common task in coding, Python eventually sprouted a new expression—the *list comprehension*—a device inspired by languages like Haskell, which is even more flexible than the original functional toolset.

As we've seen, list comprehensions apply an arbitrary *expression* to items in an iterable, rather than applying a function. Accordingly, they can be more general tools. In later Pythons, the list comprehension was extended to other roles—sets, dictionaries, and even the value generator expressions we'll explore in this chapter. Hence, it's not just for lists anymore, and we'll simply call it *comprehension* when referring to all its forms collectively.

We first met comprehensions in Chapter 4's preview, introduced them in Part II's coverage of sets, lists, and dictionaries, and studied them further in Chapter 14, in conjunction with looping statements. Because they're also related to functional programming tools like the `map` and `filter` calls and repurposed by generator expressions, though, let's resurrect the topic here for one last look.

List Comprehensions Review

Here's a brief example to refresh a few neurons. As we learned in Chapter 7, Python's built-in `ord` function returns the integer code point of a single character. If we wish to collect code points of *all* characters in an entire string, a straightforward approach is to use a simple `for` loop and append the results to a list. In a REPL:

```
>>> res = []
>>> for x in 'text':
        res.append(ord(x))                  # Manual res

>>> res
[116, 101, 120, 116]
```

Now that we know about `map`, though, we can achieve similar results with a single function call without having to manage list construction in the code:

```
>>> res = list(map(ord, 'text'))            # Apply func
>>> res
[116, 101, 120, 116]
```

However, we can get the same results from a list comprehension expression—while `map` maps a *function* over an iterable, list comprehensions map an *expression*:

```
>>> res = [ord(x) for x in 'text']          # Apply expr
>>> res
[116, 101, 120, 116]
```

List comprehensions collect the results of applying an expression to an iterable of values, and return them in a new list. Syntactically, list comprehensions are enclosed in square brackets—to remind you that they construct lists. In their basic form, within the brackets you code an expression that names a variable, followed by what looks like a `for` loop header that names the same variable. You get back the expression's results for each iteration of the implied loop.

The effect of the preceding example is similar to that of the manual `for` loop and the `map` call. List comprehensions become more convenient, though, when we wish to apply an arbitrary expression instead of a function:

```
>>> [x ** 2 for x in range(10)]             # Squares of
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

To do similar work with a `map` call, we may need to invent a little function to implement the square operation. Because we won't need this function elsewhere, we'd typically (but not necessarily) code it inline, with a `lambda`:

```
>>> list(map((lambda x: x ** 2), range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This does the same job, and it's only a few keystrokes longer than the equivalent list comprehension. It's also only marginally more complex (at

least, once you understand the `lambda`). For more advanced roles, though, list comprehensions will often require considerably less typing.

For instance, as we learned in Chapter 14, you can code an `if` clause after the comprehension's `for` to add selection logic. List comprehensions with `if` clauses can be thought of as analogous to the `filter` built-in in the preceding chapter—they skip an iterable's items for which the `if` clause is not true. As a demo, following are both schemes, along with the equivalent `for`, picking up even numbers from 0 to 9 (`x % 2` is zero for evens only):

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]

>>> list(filter((lambda x: x % 2 == 0), range(10)))
[0, 2, 4, 6, 8]

>>> res = []
>>> for x in range(10):
        if x % 2 == 0:
            res.append(x)

>>> res
[0, 2, 4, 6, 8]
```

Though it's penalized by having to code a function to be applied, the `filter` call here is still not much longer than the list comprehension either. However, the list comprehension can *combine* an `if` clause and an arbitrary expression, to give it the effect of a `filter` *and* a `map`—in a single expression:

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

This time, we collect the squares of the even numbers from 0 through 9: the `for` loop skips numbers for which the attached `if` clause on the right is false, and the expression on the left computes the squares. The equivalent `map` call would require a lot more work on our part—we would have to combine `filter` selections with `map` iteration, making for a noticeably more complex expression:

```
>>> list( map((lambda x: x**2), filter((lambda x: x % 2
[0, 4, 16, 36, 64]
```

Formal Comprehension Syntax

In fact, list comprehensions are more general still. In their simplest form, you must always code an accumulation expression and a single `for` clause:

```
[ expression for target in iterable ]
```

Though all other parts are optional, they allow richer iterations to be expressed—you can code any number of nested `for` loops in a list comprehension, and each may have an optional associated `if` test to act as a filter. The general structure of list comprehensions looks like this:

```
[ expression for target1 in iterable1 if condition1
              for target2 in iterable2 if condition2 ...
              for targetN in iterableN if conditionN ]
```

This exact same syntax is inherited by *set* and *dictionary* comprehensions as well as the *generator expressions* coming up, though these use different enclosing characters—curly braces for the first two and often-optional parentheses for the latter—and the dictionary comprehension begins with two expressions separated by a colon (for key and value).

We experimented with the `if` filter clause in the previous section. When `for` clauses are *nested* within a list comprehension, they work like equivalent nested `for` loop statements. For example:

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200,
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

This has the same effect as this substantially more verbose equivalent:

```
>>> res = []
>>> for x in [0, 1, 2]:
        for y in [100, 200, 300]:
            res.append(x + y)

>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Although list comprehensions construct list results, remember that they can iterate over any iterable object. Here's a similar bit of code that traverses strings instead of lists of numbers, and so collects concatenation results:

```
>>> [x + y for x in 'orm' for y in 'ORM']
['oO', 'oR', 'oM', 'rO', 'rR', 'rM', 'mO', 'mR', 'mM']
```

Each `for` clause can have an associated `if` filter, no matter how deeply the loops are nested—though use cases for the following sort of code, apart perhaps from the multidimensional arrays ahead, start to become more and more difficult to imagine at this level:

```
>>> [x + y for x in 'orm' if x in 'ro' for y in 'ORM' i
['oO', 'oR', 'rO', 'rR']

>>> [x + y + z for x in 'hack' if x > 'c'
               for y in 'CODE' if y in 'OD'
               for z in '123'  if z > '1']
['hO2', 'hO3', 'hD2', 'hD3', 'kO2', 'kO3', 'kD2', 'kD3'
```

Finally, here is a similar list comprehension that illustrates the effect of attached `if` selections on nested `for` clauses applied to numeric objects rather than strings:

```
>>> [(x, y) for x in range(5) if x % 2 == 0
                for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

This expression combines *even* numbers from 0 through 4 with *odd* numbers from 0 through 4. The `if` clauses filter out items in each iteration. Here is the

equivalent statement-based code:

```
>>> res = []
>>> for x in range(5):
        if x % 2 == 0:
            for y in range(5):
                if y % 2 == 1:
                    res.append((x, y))

>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Recall that if you're confused about what a complex list comprehension does, you can always nest the list comprehension's `for` and `if` clauses inside each other like this—indenting each clause successively further to the right—to derive the equivalent statements. The result is longer, but perhaps clearer in intent to some human readers on first glance, especially those more familiar with basic statements.

The `map` and `filter` equivalent of this last example would be wildly complex and deeply nested, so this book will leave its coding as an exercise for Zen masters, ex–Lisp programmers, and those with far too much free time.

Example: List Comprehensions and Matrixes

Not all list comprehensions are so artificial, of course. Let's look at one more application to stretch our comprehension muscles. As we saw in Chapters 4 and 8, one basic way to code matrixes (a.k.a. multidimensional arrays) in Python is with nested list structures. The following, for example, defines two 3 × 3 matrixes as lists of nested lists:

```
>>> M = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]

>>> N = [[2, 2, 2],
         [3, 3, 3],
         [4, 4, 4]]
```

Given this structure, we can always index rows, and columns within rows, using normal index operations:

```
>>> M[1]               # Row 2
[4, 5, 6]

>>> M[1][2]            # Row 2, item 3
6
```

List comprehensions are powerful tools for processing such structures because they automatically scan rows and columns. For instance, although this code stores the matrix by rows, to collect the second *column* we can simply iterate across the rows and pull out the desired column, or iterate through positions in the rows and index as we go:

```
>>> [row[1] for row in M]                          # Co
[2, 5, 8]

>>> [M[row][1] for row in (0, 1, 2)]               # Us
[2, 5, 8]
```

Given positions, we can also easily perform tasks such as pulling out a *diagonal*. The first of the following expressions uses `range` to generate the list of offsets and then indexes with the row and column the same, picking out `M[0][0]`, then `M[1][1]`, and so on. The second scales the column index to fetch `M[0][2]`, `M[1][1]`, etc. (we assume the matrix has the same number of rows and columns):

```
>>> [M[i][i] for i in range(len(M))]               # Di
[1, 5, 9]

>>> [M[i][len(M)-1-i] for i in range(len(M))]
[3, 5, 7]
```

Changing such a matrix *in place* requires assignment to offsets (use `range` twice if shapes differ):

```
>>> L = [[1, 2, 3], [4, 5, 6]]
>>> for i in range(len(L)):
```

```
        for j in range(len(L[i])):              # Up
            L[i][j] += 10

>>> L
[[11, 12, 13], [14, 15, 16]]
```

We can't really do the same with list comprehensions, as they make *new lists*, but we could always assign their results to the original name for a similar effect. For example, we can apply an operation to every item in a matrix, producing results in either a simple vector or a matrix of the same shape:

```
>>> [col + 10 for row in M for col in row]      # As
[11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> [[col + 10 for col in row] for row in M]
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

To understand these, translate to their simple statement form equivalents that follow—indent parts that are further to the right in the expression (as in the first loop in the following), and make a new list when comprehensions are nested on the left (like the second loop in the following). As its statement equivalent makes clearer, the second expression in the preceding works because the row iteration is an outer loop: for each row, it runs the nested column iteration to build up one row of the result matrix:

```
>>> res = []
>>> for row in M:                               # St
        for col in row:                         # In
            res.append(col + 10)

>>> res
[11, 12, 13, 14, 15, 16, 17, 18, 19]

>>> res = []
>>> for row in M:
        tmp = []                                # Le
        for col in row:
            tmp.append(col + 10)
        res.append(tmp)
```

```
>>> res
[[11, 12, 13], [14, 15, 16], [17, 18, 19]]
```

Finally, with a bit of creativity, we can also use list comprehensions to combine values of *multiple matrixes*. The following first builds a flat list that contains the result of multiplying the matrixes pairwise, and then builds a nested list structure having the same values by nesting list comprehensions again:

```
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> N
[[2, 2, 2], [3, 3, 3], [4, 4, 4]]

>>> [M[row][col] * N[row][col] for row in range(3) for
[2, 4, 6, 12, 15, 18, 28, 32, 36]

>>> [[M[row][col] * N[row][col] for col in range(3)] fc
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

This last expression works because the row iteration is an outer loop again; it's equivalent to this statement-based code:

```
res = []
for row in range(3):
    tmp = []
    for col in range(3):
        tmp.append(M[row][col] * N[row][col])
    res.append(tmp)
```

And for more fun, we can use `zip` to pair items to be multiplied—the following comprehension and loop statement both produce the same list-of-lists pairwise multiplication result as the last preceding example (and because `zip` is a generator of values, this isn't as inefficient as it may seem):

```
[[col1 * col2 for (col1, col2) in zip(row1, row2)] for

res = []
for (row1, row2) in zip(M, N):
    tmp = []
    for (col1, col2) in zip(row1, row2):
```

```
            tmp.append(col1 * col2)
        res.append(tmp)
```

Compared to their statement equivalents, the list comprehension versions here require only one line of code, might run substantially faster for large matrixes, and just might make your head explode. Which brings us to the next section.

When not to use list comprehensions: Code obfuscation

With such generality, list comprehensions can quickly become, well, incomprehensible, especially when nested. Some programming tasks are inherently complex, and we can't sugarcoat them to make them any simpler than they are (see the upcoming permutations for a prime example). Tools like comprehensions are powerful solutions when used wisely, and there's nothing inherently wrong with using them in your scripts.

At the same time, code like that at the end of the prior section may introduce more complexity than it should—and, frankly, tends to disproportionately spark the interest of those holding the darker and misinformed assumption that code obfuscation somehow implies talent. In the interest of software citizenship, some perspective seems in order here.

This book demonstrates advanced comprehensions to teach, but in the real world, programming is not about being *clever and obscure*—it's about how clearly your program communicates its purpose. Writing tricky comprehensions may be a fun academic recreation, but it doesn't work in programs that others will someday need to understand.

In other words, the age-old acronym *KISS* applies here as always: Keep It Simple—traditionally followed either by a word that is now too sexist, or another that is too colorful for a G-rated book like this. If you have to translate code to simpler statements to understand it, it should probably be simpler statements in the first place.

When to use list comprehensions: Speed, conciseness, etc.

Nevertheless, in this case, there is currently a *performance* advantage to the extra complexity: based on tests run under Python today, `map` calls and list comprehensions can be significantly faster than equivalent `for` loops. This speed difference can vary per usage and Python, but is due to the fact that

`map` and list comprehensions run at compiled-language speed inside the interpreter, which is generally faster than running `for` loop bytecode within the PVM.

Also in the plus column, list comprehensions offer a code *conciseness* that's compelling and even warranted when that reduction in size doesn't also imply a reduction in meaning for the next programmer; many find the *expressiveness* of comprehensions to be a powerful ally; and because `map` and list comprehensions are both expressions, they also can show up syntactically in places that `for` loop statements cannot, such as in `lambda` and object literals.

Because of all this, list comprehensions and `map` calls are worth knowing and using for simple sorts of iterations, especially if your application's speed is an important consideration. For example, it's hard to argue with the ease and power of code like either of the following:

```
[line.rstrip() for line in open('myfile')]
map((lambda line: line.rstrip()), open('myfile'))
```

To achieve the same memory economy and execution time division as `map`, though, list comprehensions must be coded as *generator expressions*—which is why we've run through this review in the first place, and one of the major topics this chapter turns to next.

Generator Functions and Expressions

Python today supports procrastination much more than it did in the past—it provides tools that produce results only when needed, instead of all at once. We've seen this at work in built-in tools: files that read lines on request, and functions like `map` and `zip` that produce items on demand. Such laziness isn't confined to Python itself, though. In particular, two language constructs delay result creation whenever possible in user-defined operations:

- *Generator functions* are coded as normal `def` statements, but use `yield` statements to return results one at a time, suspending and resuming their state between each.
- *Generator expressions* are similar to the list comprehensions of the prior section, but they return an object that produces results on demand instead

of building a result list.

Because neither of these constructs a result list all at once, they both save memory space and allow computation to be split across requests to avoid pauses and enable large (and even "infinite") results. As you'll see, both of these ultimately perform their delayed-results magic by implementing the *iteration protocol* we studied in Chapter 14.

These features date back to at least Python 2.2 (and were explored even earlier in languages like Icon), and are common in Python code today. Though they may initially seem unusual if you're accustomed to simpler models, you'll probably find generators to be a powerful tool. Moreover, because they are a natural extension to the function, comprehension, and iteration topics we've already explored, you already know more about coding generators than you might expect.

Generator Functions: yield Versus return

In this part of the book, we've learned about coding normal functions that receive input parameters and send back a single result immediately. It is also possible, however, to write functions that may send back a value and later be resumed, picking up where they left off. Such functions are known as *generator functions* because they generate a series of values over time instead of stopping after just one.

Generator functions are like normal functions in most respects, and in fact are coded with normal `def` statements. However, when created, they are compiled specially into an object that supports the iteration protocol. When called, they don't return a result: they return a result generator that can appear in any iteration context. We studied iterables in Chapter 14, and Figure 14-1 gave a formal and graphic summary of their operation. Here, we'll revisit the subject to see how it applies to generators.

State suspension

Unlike normal functions that return a value and exit, generator functions automatically suspend and resume their execution and state around the point of value generation. Because of that, they are often a useful alternative to both computing an entire series of values up front and manually saving and restoring state in classes. The *state* that generator functions retain when they

are suspended includes both their code location and their entire local scope. Hence, their *local variables* retain information between results, and make it available when the functions are resumed.

The chief code difference between generator and normal functions is that a generator *yields* a value, rather than *returning* one—the `yield` statement suspends the function and sends a value back to the caller, but retains enough state to enable the function to resume from where it left off. When resumed, the function continues execution immediately after the last `yield` run. From the function's perspective, this allows its code to produce a series of values over time, rather than computing them all at once and sending them back in something like a list.

Iteration protocol integration

To truly understand generator functions, you need to know that they are closely bound up with the notion of the iteration protocol in Python. As we've seen, iterator objects define a `__next__` method, which either returns the next item in the iteration, or raises the `StopIteration` exception to end the iteration. An iterable object's iterator is fetched initially with the `__iter__` method, though this step is a no-op for objects that are their own iterator.

Python `for` loops, and all other iteration tools, use this iteration protocol to step through a sequence or value generator, if the protocol is supported (if not, iteration falls back on repeatedly indexing sequences instead). Any object that supports this interface works in all iteration tools, and the `iter` and `next` built-ins simplify manual iteration by calling the corresponding double-underscore method (or its internal equivalent).

To support this protocol, functions containing a `yield` statement are compiled specially as *generators*—they are not normal functions, but rather are built to return an object with the expected iteration-protocol methods. When later called, such functions return an iterable object that supports the iteration protocol with an automatically created method named `__next__` to start or resume execution.

Besides `yield`, generator functions may also use a `return` statement that, along with falling off the end of the `def` block, terminates the generation of values by automatically raising a `StopIteration` exception. A generator's `return` can also give an object that becomes the `value` attribute of the

`StopIteration` exception, but it's ignored by iteration tools and uncommon. From the caller's perspective, the generator's `__next__` method simply starts or resumes the function and runs until either the next `yield` result is returned, or a `StopIteration` is raised.

The net effect is that generator functions, coded as `def` statements containing `yield` statements, are automatically made to support the iteration methods protocol and thus may be used in any iteration tool to produce results over time and on demand. The presence of a `yield` in a `def` suffices to make all this happen, and none of this applies to `lambda` because it doesn't allow statements like `yield` (which is yet another reason to prefer `def`).

Generator functions in action

As usual, this is probably simpler in code than narrative. The following defines a generator function that can be used to produce the squares of a series of numbers—over time:

```
>>> def gensquares(n):
        for i in range(n):
            yield i ** 2        # Resume here later
```

This function yields a value, and so returns to its caller, each time through the loop. When it is resumed, its prior state is restored, including the last values of its variables `i` and `n`, and control picks up again immediately after the `yield` statement. For example, when it's used in the body of a `for` loop, the first iteration starts the function and gets its first result; thereafter, control returns to the function after its `yield` statement each time through the loop:

```
>>> for i in gensquares(5):     # Resume the function
        print(i, end=' ')       # Print last yielded vc

0 1 4 9 16
```

To end the generation of values, functions either use a `return` statement or simply allow control to fall off the end of the function body. As noted, `return` can give a value attached to the exit exception, but usually does not.

To most people, this process seems a bit implicit (if not enchanted) on first encounter. It's actually quite tangible, though. If you really want to see what is going on inside the `for`, call the generator function directly:

```
>>> x = gensquares(3)           # Generator of 0..2 squ
>>> x
<generator object gensquares at 0x10dc6d700>
```

You get back a *generator object* that supports the iteration protocol—the generator function was compiled to return this automatically. The returned generator object in turn has a `__next__` method that starts the function or resumes it from where it last yielded a value, and raises a `StopIteration` exception when the end of the series of values is reached and the function returns. As noted, `next(X)` here calls `X.__next__()` for convenience:

```
>>> next(x)                     # Run to first yield
0
>>> next(x)                     # Resume after yield, r
1
>>> next(x)                     # Resume after yield, r
4
>>> next(x)                     # Resume after yield, e
StopIteration
```

Per Chapter 14, `for` loops and other iteration tools work with generators in the same way—by calling the `__next__` method repeatedly, until the exit exception is caught. For generators, the result produces yielded values over time.

Notice that the top-level `iter` call of the iteration protocol isn't required here because generators are their own iterator, supporting just one active iteration scan. To put that another way, generators return themselves for `iter` and support `next` directly. This also holds true in the generator expressions you'll meet later in this chapter:

```
>>> y = gensquares(5)           # Returns a generator w
>>> iter(y) is y                # iter() is not require
True
```

```
>>> next(y)                     # Can run next() immedi
0
```

Why generator functions?

Given the simple example we're using to illustrate fundamentals, you might be wondering just why you'd ever care to code a generator at all. In code so far, for instance, we could simply build the list of result values all at once:

```
>>> def buildsquares(n):
        res = []
        for i in range(n): res.append(i ** 2)
        return res

>>> for x in buildsquares(5): print(x, end=' ')

0 1 4 9 16
```

For that matter, we could use any of the `for` loop, `map`, or list comprehension techniques we've already mastered:

```
>>> for x in [n ** 2 for n in range(5)]:
        print(x, end=' ')

0 1 4 9 16

>>> for x in map((lambda n: n ** 2), range(5)):
        print(x, end=' ')

0 1 4 9 16
```

However, generators can be better in terms of both memory use and performance in larger programs. They allow functions to avoid doing all the work up front, which is especially useful when the result lists are large or when it takes a lot of computation to produce each value. Generators distribute the time required to produce the series of values among loop iterations, and can even produce a series of values that has no end at all (an "infinite" result).

Moreover, for more advanced uses, generators can provide a simpler alternative to manually saving the state between iterations in class objects—with generators, variables accessible in the function's scopes are saved and restored automatically. You'll be able to judge this contrast after we discuss class-based iterables in more detail in Part VI.

Generator functions are also more broadly focused than implied so far. They can operate on and return any type of object, and as *iterables* may appear in any of Chapter 14's iteration tools, including `tuple` calls, enumerations, and dictionary comprehensions:

```
>>> def ups(line):
        for sub in line.split(','):          # Substrir
            yield sub.upper()

>>> tuple(ups('aaa,bbb,ccc'))               # All iter
('AAA', 'BBB', 'CCC')

>>> {i: s for (i, s) in enumerate(ups('aaa,bbb,ccc'))}
{0: 'AAA', 1: 'BBB', 2: 'CCC'}
```

In a moment you'll observe the same assets for generator expressions—a tool that trades function flexibility for comprehension conciseness. Later in this chapter you'll also see that generators can sometimes enable otherwise impractical tasks, by producing components of result sets that would be far too large to create all at once. First, though, let's explore some advanced generator function features.

Extended generator function protocol: send versus next

Somewhere along generators' evolutionary path (in Python 2.5), a `send` method was added to the generator function protocol. The `send` method advances to the next item in the series of results, just like `__next__`, but also provides a way for the caller to communicate with the generator, and hence to affect its operation.

Technically, `yield` is an *expression* form that returns the item passed to `send`, not a statement. It can be coded either way (and usually is a statement), but when used as an expression must be enclosed in parentheses

unless it's the only item on the right side of an assignment statement. For example, `X = yield Y` is OK, as is `X = (yield Y) + Z`.

When this extra protocol is used, values are sent into a generator `G` by calling `G.send(value)`. The generator's code is then resumed, and the `yield` expression inside the generator returns the value passed to `send` by the caller. If the regular `G.__next__()` method (or its `next(G)` equivalent) is called to advance, the `yield` simply returns `None`. For example:

```
>>> def gen():
        for i in range(10):
            X = yield i
            print('=>', X)

>>> G = gen()
>>> next(G)                 # Must call next() first, to s
0
>>> G.send(77)              # Advance, and send value to y
=> 77
1
>>> G.send(88)
=> 88
2
>>> next(G)                 # next() and X.__next__() send
=> None
3
```

The `send` method can be used, for example, to code a generator that its caller can terminate by sending a termination code, or redirect by passing a new position in data being processed inside the generator.

In addition, generators also support a `throw(type)` method to raise an exception inside the generator at the latest `yield`, and a `close` method that raises a special `GeneratorExit` exception inside the generator to terminate the iteration entirely. Together with `send`, these are advanced features added to make generators more like a tool called coroutines, a role eventually subsumed in part by the upcoming `async`. Hence, we won't delve further here; see Python's standard manuals for more information, and watch for more on exceptions in Part VII.

Note, though, that while Python provides a `next(X)` convenience built-in that calls the `X.__next__()` method of an object, other generator methods, like `send`, must be called as methods of generator objects directly (e.g., `G.send(X)`). This makes sense if you realize that these extra methods are implemented on built-in generator objects only, whereas the `__next__` method applies to all iterable objects—both built-in types and user-defined classes.

The yield from extension

Even later, Python 3.3 introduced extended syntax for the `yield` statement with a `from *generator*` clause that allows generators to delegate to nested generators (known as *subgenerators*). In simple cases, it's the equivalent to a yielding `for` loop. As a demo, the `list` call in the following forces a generator to produce values from two sources:

```
>>> def both(N):
        for i in range(N): yield i
        for i in map(lambda x: x ** 2, range(N)): yield

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]
```

The `yield from` syntax makes this more concise and explicit, and supports all the usual generator usage contexts. In the following, `both` is called a *delegating* generator, and the `range` and `map` built-ins serve as subgenerators (`join` also uses a generator expression here: see the next section):

```
>>> def both(N):
        yield from range(N)
        yield from map(lambda x: x ** 2, range(N))

>>> list(both(5))
[0, 1, 2, 3, 4, 0, 1, 4, 9, 16]

>>> ' : '.join(str(i) for i in both(5))
'0 : 1 : 2 : 3 : 4 : 0 : 1 : 4 : 9 : 16'
```

The `yield from` also supports arbitrary *chains* of generators. In the following, for example, two generators `yield from` others, the last of which ultimately uses `yield` to send results up though the chain:

```
>>> def c(n):
        yield n * 8                    # The bottom of t
        yield n ** 2

>>> def b(n):
        yield from c(n)                # Delegate to sub

>>> def a(n):
        yield from b(n)                # Delegate to sub

>>> g = a(4)                           # Make top generc
>>> g
<generator object a at 0x10bd17940>
>>> next(g)
32

>>> for i in a(4): print(i)           # Force results p
32
16
```

In more advanced roles, though, this extension allows subgenerators to receive *sent* and *thrown* values directly from the delegating generator's caller, and return a final value to the delegating generator as the result of the `yield from` expression. The net effect allows generators to be split into multiple subgenerators much as a single function can be split into multiple subfunctions, but still operate as though their code appeared inline where the `yield from` appears.

Since this is both uncommon and beyond this chapter's scope, we'll again defer to Python's standard manuals for more details. For an additional `yield from` example, see the solution to Exercise 11 from "Test Your Knowledge: Part IV Exercises", and stay tuned for a glimpse of `async def` coroutines ahead—whose `await` is something like a generator `yield from`, with a delegation to an event loop to allow other tasks to be run during pauses. Here, let's move on to a tool close enough to `yield` to be called a fraternal twin.

Generator Expressions: Iterables Meet Comprehensions

Because the delayed evaluation of generator functions was so useful, later Pythons eventually combined the notions of iterables and comprehensions in a new tool: *generator expressions*. Syntactically, generator expressions are just like normal list comprehensions, and support all their syntax—including `if` filters and `for` loop nesting—but they are enclosed in parentheses instead of square brackets (like tuples, their enclosing parentheses are often optional):

```
>>> [x ** 2 for x in range(5)]          # List comprehe
[0, 1, 4, 9, 16]

>>> (x ** 2 for x in range(5))          # Generator exp
<generator object <genexpr> at 0x109607ac0>
```

In fact, at least on a functionality basis, coding a list comprehension is essentially the same as wrapping a generator expression in a `list` built-in call to force it to produce all its results in a list at once:

```
>>> list(x ** 2 for x in range(5))      # List comprehe
[0, 1, 4, 9, 16]
```

Operationally, however, generator expressions are very different: instead of building the result list in memory, they return a *generator object*—an automatically created iterable. This iterable object in turn supports the *iteration protocol* to yield one piece of the result list at a time in any iteration tool. The iterable object also retains generator state while active—the variable `x` in the preceding expressions, along with the generator's code location.

The net effect is much like that of generator functions, but in the context of a comprehension *expression*: we get back an object that remembers where it left off after each part of its result is returned. Also like generator functions, looking under the hood at the protocol that these objects automatically support can help demystify them; the `iter` call is again not required at the top here, for reasons we'll expand on ahead:

```
>>> G = (x ** 2 for x in range(3))
>>> iter(G) is G                        # iter(G) optic
True
>>> next(G)                             # Generator obj
0
>>> next(G)
1
>>> next(G)
4
>>> next(G)
StopIteration

>>> G
<generator object <genexpr> at 0x109607920>
```

Again, we don't typically see the `next` iterator machinery under the hood of a generator expression like this because `for` loops and similar tools trigger it for us automatically:

```
>>> for num in (x ** 2 for x in range(3)):         # Cal
        print(num, num / 2.0, sep=', ')

0, 0.0
1, 0.5
4, 2.0
```

As we've already seen, every iteration tool does this—including `for` loops; the `list` call we used earlier; comprehensions of all kinds; the `map` and `filter` functional tools; and other iteration tools we explored in Chapter 14, including the `sum`, `sorted`, `any`, and `all` built-in functions. As *iterables*, generator expressions can be used in any of these iteration tools, just like both physical sequences and the results of a generator-function call.

For example, the following deploys generator expressions in the string `join` method call and tuple assignment, iteration tools both. Recall that `join` with an empty-string delimiter concatenates values produced by its iterable:

```
>>> ''.join(x.upper() for x in 'aaa,bbb,ccc'.split(',')
'AAABBBCCC'
```

```
>>> a, b, c = (x + '\n' for x in 'aaa,bbb,ccc'.split(',
>>> a, c
('aaa\n', 'ccc\n')
```

Notice how the `join` call in the preceding doesn't require *extra* parentheses around the generator. Syntactically, parentheses are *not required* around a generator expression that is the sole item already enclosed in parentheses used for other purposes—like those of a function call. Parentheses are required in all other cases, however, even if they seem extra, as in the second call to `sorted` that follows:

```
>>> sum(x ** 2 for x in range(3))
5
>>> sorted(x ** 2 for x in range(3))
[0, 1, 4]
>>> sorted((x ** 2 for x in range(3)), reverse=True)
[4, 1, 0]
```

Like the often-optional parentheses in tuples, there is no widely accepted rule on this, though a generator expression does not have as clear a role as a fixed collection of other objects as a tuple, making extra parentheses seem perhaps more spurious here.

Why generator expressions?

Just like generator functions, generator expressions are a *memory-space* optimization—they do not require the entire result list to be constructed all at once, as the square-bracketed list comprehension does. Also like generator functions, they divide the work of results production into smaller *time slices*—they yield results in piecemeal fashion, instead of making the caller wait for the full set to be created in a single call.

On the other hand, because generator expressions often run *slower* today than list comprehensions, they may be best reserved for result sets that are very large, or programs that cannot wait for full results generation. A more authoritative statement about performance, though, will have to await the timing scripts we'll code in the next chapter.

On the subjective upside, generator expressions also offer significant *coding* advantages—as the next sections show.

Generator expressions versus map

One way to see the coding benefits of generator expressions is to compare them to other functional tools, as we did for list comprehensions. For example, generator expressions often are equivalent to `map` calls, because both generate result items on request. Like list comprehensions, though, generator expressions may be simpler to code when the operation applied is not a function call:

```
>>> list(map(abs, (-1, -2, 3, 4)))
[1, 2, 3, 4]
>>> list(abs(x) for x in (-1, -2, 3, 4))
[1, 2, 3, 4]

>>> list(map(lambda x: x * 2, (1, 2, 3, 4)))
[2, 4, 6, 8]
>>> list(x * 2 for x in (1, 2, 3, 4))
[2, 4, 6, 8]
```

The same holds true for text-processing use cases like the `join` call we saw earlier—a list comprehension makes an extra temporary list of results, which is completely *pointless* in this context because the list is not retained, and `map` loses simplicity points compared to generator expression syntax when the operation being applied is not a call:

```
>>> line = 'aaa,bbb,ccc'
>>> ''.join([x.upper() for x in line.split(',')])
'AAABBBCCC'

>>> ''.join(x.upper() for x in line.split(','))
'AAABBBCCC'
>>> ''.join(map(str.upper, line.split(',')))
'AAABBBCCC'

>>> ''.join(x * 2 for x in line.split(','))
'aaaaaabbbbbbcccccc'
>>> ''.join(map(lambda x: x * 2, line.split(',')))
'aaaaaabbbbbbcccccc'
```

Both `map` and generator expressions can also be arbitrarily *nested*; when coded this way, a `list` call or other iteration tool starts the entire process of producing results. For example, the list comprehension in the following produces the same result as the `map` and generator equivalents that follow it, but makes two physical lists. The others generate just one integer at a time with nested generators, and the generator expression form may more clearly reflect its intent:

```
>>> [x * 2 for x in [abs(x) for x in (-1, -2, 3, 4)]]
[2, 4, 6, 8]

>>> list(map(lambda x: x * 2, map(abs, (-1, -2, 3, 4))))
[2, 4, 6, 8]

>>> list(x * 2 for x in (abs(x) for x in (-1, -2, 3, 4)))
[2, 4, 6, 8]
```

Although the effect of all three of these is to combine operations, the generators do so without making multiple temporary lists. In fact, the next example both nests *and* combines generators—the nested generator expression is activated by `map` , which in turn is only activated by `list` :

```
>>> import math
>>> list(map(math.sqrt, (x ** 2 for x in range(4))))
[0.0, 1.0, 2.0, 3.0]
```

Technically speaking, the `range` on the right in the preceding is a value generator too, activated by the generator expression itself—forming *three levels* of value generation, which produce individual values from inner to outer only on request, and which "just works" because of Python's iteration tools and protocol. In fact, generator nestings can be arbitrarily mixed and deep, though some may be more valid than others:

```
>>> list(map(abs, map(abs, map(abs, (-1, 0, 1)))))
[1, 0, 1]
>>> list(abs(x) for x in (abs(x) for x in (abs(x) for x
[1, 0, 1]
```

These last examples illustrate how general generators can be, but are also coded in an intentionally complex form to underscore that generator expressions have the same potential for abuse as the list comprehensions discussed earlier—as usual, you should keep them simple unless they must be complex.

When used well, though, generator expressions combine the expressiveness of list comprehensions with the space and time benefits of other iterables. The following *nonnested* alternatives, for example, provide simpler solutions to the preceding three listings' nested codings, but still leverage generators' strengths:

```
>>> list(abs(x) * 2 for x in (-1, -2, 3, 4))
[2, 4, 6, 8]
>>> list(math.sqrt(x ** 2) for x in range(4))
[0.0, 1.0, 2.0, 3.0]
>>> list(abs(x) for x in (-1, 0, 1))
[1, 0, 1]
```

Generator expressions versus filter

Generator expressions also support all the usual list comprehension syntax—including `if` clauses, which work like the `filter` call we met earlier. Because `filter` is an iterable that generates its results on request, a generator expression with an `if` clause is operationally equivalent. Again, the `join` in the following is an iteration tool that suffices to force all forms to produce their results:

```
>>> line = 'aa bbb c'
>>> ''.join(x for x in line.split() if len(x) > 1)
'aabbb'
>>> ''.join(filter(lambda x: len(x) > 1, line.split()))
'aabbb'
```

The generator seems just marginally simpler than the `filter` here. As for list comprehensions, though, adding processing steps to `filter` results requires a `map` too, which makes `filter` noticeably more complex:

```
>>> ''.join(x.upper() for x in line.split() if len(x) >
'AABBB'
>>> ''.join(map(str.upper, filter(lambda x: len(x) > 1,
'AABBB'
```

In effect, generator expressions provide more general coding structures that do not rely on functions, but still delay results production. Also like list comprehensions, there is always a statement-based equivalent to a generator expression, though it sometimes renders substantially more code:

```
>>> res = ''
>>> for x in line.split():
        if len(x) > 1:
            res += x.upper()

>>> res
'AABBB'
```

In this case, though, the statement form isn't quite the same—it cannot produce items one at a time, and it's also emulating the effect of the `join` that forces results to be produced all at once. The true equivalent to a generator expression would be a generator function with a `yield`, as the next section will show.

Generator Functions Versus Generator Expressions

Let's recap what we've covered so far in this section:

Generator functions

A function `def` statement that contains a `yield` statement is turned into a generator function. When called, it returns a new *generator object* with automatic retention of local scope and code position; an automatically created `__iter__` method that simply returns itself; and an automatically created `__next__` method that starts the function or resumes it where it last left off, and raises `StopIteration` when finished producing results.

Generator expressions

A comprehension expression enclosed in parentheses is known as a generator expression. When run, it returns a new *generator object* with the same automatically created method interface and state retention as a generator function call's results—with an `__iter__` method that simply returns itself; and a `__next__` method that starts the implied loop or resumes it where it last left off, and raises `StopIteration` when finished producing results.

The net effect is to automatically produce results on demand in all iteration tools that employ either of these.

As implied by preceding coverage, the same iteration can often be coded with *either* a generator function or a generator expression. The following generator *expression*, for example, repeats each character in a string four times:

```
>>> G = (c * 4 for c in 'hack')           # Generator e
>>> list(G)                               # Force gener
['hhhh', 'aaaa', 'cccc', 'kkkk']
```

The equivalent generator *function* requires slightly more code, but its multiple-statement block will be able to code more logic and use more state information if needed. In fact, this is essentially the same as the prior chapter's trade-off between `lambda` and `def` —expression conciseness versus statement power:

```
>>> def timesfour(S):                     # Generator f
        for c in S:
            yield c * 4

>>> G = timesfour('hack')
>>> list(G)                               # Iterate aut
['hhhh', 'aaaa', 'cccc', 'kkkk']
```

To their clients, the two are more similar than different. For instance, both expressions and functions support both automatic and manual iteration—the prior `list` call iterates automatically, and the following iterate manually:

```
>>> G = (c * 4 for c in 'hack')
>>> I = iter(G)                           # Iterate mar
```

```
>>> next(I)
'hhhh'
>>> next(I)
'aaaa'

>>> G = timesfour('hack')
>>> I = iter(G)                              # Iterate mar
>>> next(I)
'hhhh'
>>> next(I)
'aaaa'
```

In either case, Python automatically creates a generator object, which has both the methods required by the iteration protocol, and state retention for variables in the generator's code and its current code location. Notice how we make new generators here to iterate again—as explained in the next section, generators are single-pass iterators.

First, though, here's the true statement-based equivalent of the expression at the end of the prior section: a function that yields values—though the difference is irrelevant if the code using it produces all results with a tool like `join`:

```
>>> line = 'aa bbb c'

>>> ''.join(x.upper() for x in line.split() if len(x) >
'AABBB'

>>> def gensub(line):
        for x in line.split():
            if len(x) > 1:
                yield x.upper()

>>> ''.join(gensub(line))
'AABBB'
```

While generators have valid roles, in cases like this the use of generators over the simple statement equivalent shown earlier may be difficult to justify, except on stylistic grounds: if you're just going to immediately `join` generated results anyhow, you might as well skip generators and use simple loops. On the other hand, trading four lines for the generator expression's one may to many seem fairly compelling stylistic grounds!

Generator Odds and Ends

This section wraps up generators with a quick rundown of associated but lesser topics. After this, we'll move on to larger examples, but make sure you have a handle on the smaller bits before jumping ahead.

Generators are single-pass iterables

First up, a subtle but important point: the objects returned by both generator functions and generator expressions are their own iterators and thus support just *one active iteration*—unlike some built-in types, you can't have multiple iterators of either generator positioned at different locations in the stream of results. Because of this, a generator's iterator is the generator itself; in fact, as suggested earlier, calling `iter` on a generator expression or function is an optional no-op:

```
>>> G = (c * 4 for c in 'hack')
>>> iter(G) is G                        # My iterator
True
```

If you do iterate over the results stream manually with multiple iterators, they will all point to the same position:

```
>>> G = (c * 4 for c in 'hack')         # Make a new
>>> I1 = iter(G)                        # Iterate mar
>>> next(I1)
'hhhh'
>>> next(I1)
'aaaa'
>>> I2 = iter(G)                        # Second iter
>>> next(I2)
'cccc'
```

Moreover, once any iteration runs to completion, all are exhausted—we have to make a new generator to start again:

```
>>> list(I1)                            # Collect the
['kkkk']
>>> next(I2)                            # Other iterc
```

```
StopIteration

>>> I3 = iter(G)                              # Ditto for r
>>> next(I3)
StopIteration

>>> I3 = iter(c * 4 for c in 'hack')          # New generat
>>> next(I3)
'hhhh'
```

The same holds true for generator functions—the following `def` statement-based equivalent supports just one active iterator and is exhausted after one pass:

```
>>> def timesfour(S):
        for c in S:
            yield c * 4

>>> G = timesfour('hack')                     # Generator f
>>> iter(G) is G                              # One scan pe
True
>>> I1, I2 = iter(G), iter(G)
>>> next(I1)
'hhhh'
>>> next(I1)
'aaaa'
>>> next(I2)                                  # I2 at same
'cccc'
```

This is different from the behavior of some built-in objects like lists, which support multiple iterators and passes and even reflect their in-place changes in active iterators:

```
>>> L = [1, 2, 3, 4]
>>> I1, I2 = iter(L), iter(L)
>>> next(I1)
1
>>> next(I1)
2
>>> next(I2)                                  # Lists suppo
1
>>> del L[2:]                                 # Changes ref
```

```
>>> next(I1)
StopIteration
```

Though not readily apparent in these simple examples, this can matter in your code: if you wish to scan a generator's values multiple times, you must either create a *new* generator for each scan or build a rescannable *list* out of its values—a single generator's values will be consumed and exhausted after a single pass. See this chapter's sidebar "Why You Will Care: Iteration Versus Python Morph" for a prime example of the sort of code impacted by this.

When we begin coding *class-based* iterables in Part VI, you'll also see that it's up to us to decide how many iterations we wish to support for our objects, if any. In general, objects that wish to support multiple scans will return supplemental class objects instead of themselves for `iter`. The next section previews more of this model.

Generation in built-ins and classes

Although we've focused on coding value generators ourselves in this section, don't forget that many built-in types behave the same way. As we saw in Chapter 14, for example, *dictionaries* and *files* generate results too:

```
for key in dictionary: …                    # See
for line in file: …
```

Though beyond this book's scope, many Python standard-library tools generate values too, including its *directory walker*—which at each level of a folder tree yields a tuple of the current directory, its subdirectories, and its files:

```
>>> import os
>>> for (root, subs, files) in os.walk('..'):      # Dir
        for name in files:                          # A f
            if name.endswith('.py'):                # Als
                print(root, name)

../Chapter02 script0.py
../Chapter03 myfile.py
```

```
../Chapter03 script1.py
…etc…
```

Because the current Python-coded implementation of `os.walk` uses `yield` to return results, it's a normal generator function, and hence iterable object, that generates a three-item tuple on each iteration:

```
>>> G = os.walk('..')       # A single-scan iterator: i
>>> next(G)
('..', ['Chapter02', 'Chapter03', 'Chapter04', …etc… 'C
>>> next(G)
('../Chapter02', ['__pycache__'], ['script0.py'])
```

By yielding results as it goes, the walker does not require its clients to wait for an entire tree to be scanned. See Python's manuals for more on this tool. For system-tools fans, also see the next chapter for an example that uses `os.popen` —a related iterable used to run a shell command and read its output. And for additional examples of built-in value generators and the iteration tools that use them, review Chapter 14.

Finally, although beyond the scope of this chapter, it is also possible to implement arbitrary user-defined generator objects with *classes* that conform to the iteration protocol. Such classes define `__iter__` and `__next__` methods explicitly to support the protocol:

```
class SomeIterable:
    def __iter__(...): ...      # On iter(): return self
    def __next__(...): ...      # On next(): coded here,
```

As the prior section suggested, these classes usually return their objects directly for single-iteration behavior, or a supplemental object with scan-specific state for multiple-scan support.

Alternatively, a user-defined iterable class's `__iter__` may use `yield` to transform itself into a generator, and a `__getitem__` indexing method is also available as a fallback option for iteration with trade-offs we'll explore later. However this is coded, the iterator and generator story won't really be complete until we've seen how it also maps to classes. For now, we'll have to settle for postponing its conclusion—and its final sequel—until Chapter 30.

Comprehensions versus type calls and generators

We've also been focusing on list comprehensions and generators in this chapter, but keep in mind that there are two other comprehension expression forms: the set and dictionary comprehensions we met earlier in Chapter 14 and Part II. For reference and closure, here's a summary of all the comprehension forms in Python:

```
>>> [x * x for x in range(10)]                # List com
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]          # Fixed or

>>> (x * x for x in range(10))                # Generato
<generator object <genexpr> at 0x100a1f920>

>>> {x * x for x in range(10)}                # Set comp
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}          # Random c

>>> {x: x * x for x in range(10)}             # Dict com
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8:
```

All of these forms use the same formal comprehension syntax we met earlier, but their enclosing delimiters (and key, for dictionaries) denote their differing roles. In a sense, list, set, and dictionary comprehensions are syntactic sugar for passing generator expressions to type names. Because all accept any iterable, a generator works here too:

```
>>> list(x ** 2 for x in range(10))           # Generato
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> set(x * x for x in range(10))             # Similar
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}

>>> dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8:
```

But don't read too much into such equivalences. Because implementations may vary arbitrarily, you should generally collect performance data before adopting an alternative. In this case, generators are actually slower than the equivalent list comprehension today, as we'll prove in the next chapter. Programs that run lots of them may need to care.

Scopes and comprehension variables

Now that we've seen all comprehension forms, Chapter 17's overview of the localization of loop variables in these expressions may make more sense. In all forms, the loop variable (or variables) after the `for` is local to the expression—it won't clash with names outside, but is also not available there, and this differs from `for` statements:

```
>>> X = 99
>>> [X for X in range(5)]          # All comprehensions
[0, 1, 2, 3, 4]
>>> X                              # Enclosing-scope X w
99
>>> for X in range(5): pass        # But loop statements
>>> X
4
```

As noted in Chapter 17, loop variables assigned in a comprehension really live in a further nested special-case scope, but other names referenced within these expressions follow the usual LEGB rules. In the following generator, for example, `Z` is localized in the comprehension, but `Y` and `X` are found in the enclosing local and global scopes as usual:

```
>>> X = 'aaa'
>>> def func():
        Y = 'bbb'
        print('-'.join(Z for Z in X + Y))    # Z=compre

>>> func()
a-a-a-b-b-b
```

One exception here: names assigned by the `:=` expression inside a comprehension do leak out of comprehension, and generally behave as though they were assigned in the scope containing the comprehension itself:

```
>>> S = 'hack'
>>> [(temp := S[i]) + temp.upper() for i in range(len(S
['hH', 'aA', 'cC', 'kK']
>>> temp
'k'
```

```
>>> i
NameError: name 'i' is not defined. Did you mean: 'id'?
```

While this allows comprehensions to both reuse and export values, keep in mind that a `lambda` container's local scope effectively plugs the leak—`temp` in the following, for example, lives only in the `lambda`'s scope:

```
>>> del temp
>>> f = lambda: [(temp := S[i]) + temp.upper() for i ir
>>> f()
['hH', 'aA', 'cC', 'kK']
>>> temp
NameError: name 'temp' is not defined
```

We'll code such a `:=` nesting in a sequence-scrambler example ahead, though some observers may deem it much less transparent than simple assignment statements, and other real-world roles must await your discovery. See Chapter 11 for more background on `:=` named assignment.

Generating "infinite" (well, indefinite) results

Finally, it was noted earlier in passing that generators can even produce "infinite" results that tools like list comprehensions cannot. This may sound more impressive than it is; really, this just means that a generator can keep yielding results from its retained state's local variables *indefinitely*—until its client grows tired of them:

```
>>> def squares(n):
        while True:                         # Generate res
            yield n ** 2                    # Or until no
            n += 1

>>> G = squares(2)
>>> next(G)
4
>>> next(G)
9
>>> for i in range(10): print(next(G), end=' ')

16 25 36 49 64 81 100 121 144 169
```

This pattern may be useful in limited roles like test-parameter generation, and other tools can't compete in this event; list comprehensions, for example, must collect all results at once. In the end, though, this is a narrow role, generators are mortal, and so are we—so let's move on to some code that's a bit more tangible in the next section.

Example: Shuffling Sequences

To demonstrate the power of iteration and generation tools in action, let's turn to some more complete examples. In Chapter 18, we wrote a testing function, based on earlier code in Chapter 13, that scrambled the order of arguments used to verify generalized intersection and union functions. There, it was noted that this might be better coded as a generator of values. Now that we've learned how to write generators, this serves to illustrate a practical application.

One note up front: because they slice and concatenate objects, all the examples in the section (including the permutations at the end) work only on *sequences* like strings and lists, not on arbitrary *iterables* like files, maps, and other generators. That is, some of these examples will *be* generators themselves, producing values on request, but they cannot process generators as their *inputs*. Generalization for broader categories is left as an open issue, though the code here will suffice unchanged if you wrap nonsequence generators in `list` calls before passing them in.

Scrambling Sequences

First, let's review. As coded in Chapter 18, we can reorder a sequence with slicing and concatenation, moving the front item to the end on each loop; *slicing* instead of indexing the item allows `+` to work for arbitrary sequence types:

```
>>> L, S = [1, 2, 3], 'code'

>>> for i in range(len(S)):              # Coding 1: for
        S = S[1:] + S[:1]                # Move front ite
        print(S, end=' ')

odec deco ecod code

>>> for i in range(len(L)):
```

```
        L = L[1:] + L[:1]               # Slice so any s
        print(L, end=' ')

[2, 3, 1] [3, 1, 2] [1, 2, 3]
```

Alternatively, as we also saw in Chapter 13, we get the same results by moving an entire front section to the end, though the order of the results varies slightly:

```
>>> for i in range(len(S)):             # Coding 2: for
        X = S[i:] + S[:i]               # Rear part + fr
        print(X, end=' ')

code odec deco ecod
```

Trace this code to see how it works; each version produces N results for an N-item sequence passed in.

Simple functions

As is, this code works on specific named variables only, and simply prints its result. As we've seen, it's easy to generalize this by turning it into a normal *function* that can both work on any object passed to its argument and return its result for arbitrary use. The following does so for the second coding alternative; since its first cut exhibits the classic list comprehension pattern, we can also save some work by coding it as such in the second:

```
>>> def scramble(seq):
        res = []
        for i in range(len(seq)):
            res.append(seq[i:] + seq[:i])
        return res

>>> scramble('code')
['code', 'odec', 'deco', 'ecod']

>>> def scramble(seq):
        return [seq[i:] + seq[:i] for i in range(len(se

>>> scramble('code')
['code', 'odec', 'deco', 'ecod']
```

```
>>> for x in scramble((1, 2, 3)):
        print(x, end=' ')

(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

We could use recursion here as well, but it's probably overkill in this fixed and linear context.

Generator functions

The preceding section's simple approach works, but must build an entire result list in memory all at once (not great on memory usage if it's massive), and requires the caller to wait until the entire list is complete (less than ideal if this takes a substantial amount of time). You should know by now that we can do better on both fronts by translating this to a *generator function* that yields one result at a time, using either coding scheme:

```
>>> def scramble(seq):
        for i in range(len(seq)):
            seq = seq[1:] + seq[:1]                 # Ger
            yield seq                               # Ass

>>> def scramble(seq):
        for i in range(len(seq)):                   # Ger
            yield seq[i:] + seq[:i]                 # Yie
```

Both of these alternatives produce values when used by an iteration tool like `list` or `for`, though the second version's results order (shown here) again varies slightly in ways that probably won't matter to future clients:

```
>>> list(scramble('code'))                          # lis
['code', 'odec', 'deco', 'ecod']

>>> list(scramble((1, 2, 3)))                       # Any
[(1, 2, 3), (2, 3, 1), (3, 1, 2)]

>>> for x in scramble((1, 2, 3)):                   # for
        print(x, end=' ')

(1, 2, 3) (2, 3, 1) (3, 1, 2)
```

Both generator functions retain their local scope state (`seq` and `i`) while active, minimize memory space requirements, and divide the work into shorter time slices. As full functions, they are also very general. Moreover, because iteration tools work the same whether stepping through a real list or a generator of values, the function can select between the two codings freely, and even change strategies in the future without impacting callers.

Generator expressions

As we've also seen, *generator expressions*—comprehensions in parentheses instead of square brackets—also generate values on request and retain their local state. As expressions, they're not as flexible as full functions, but because they yield their values automatically, generator expressions can often be more concise in specific use cases like this:

```
>>> S = 'code'
>>> G = (S[i:] + S[:i] for i in range(len(S)))
>>> list(G)
['code', 'odec', 'deco', 'ecod']
```

A generator expression can't use the `seq` assignment statement of the first generator-function coding, but it *can* achieve the same effect by nesting the `:=` *named-assignment* expression covered in Chapter 11, because `:=` both changes assigned names in the containing scope and returns the assigned value so it appears in the results stream:

```
>>> S = 'code'
>>> G = (S:=(S[1:] + S[:1]) for i in range(len(S)))
>>> list(G)
['odec', 'deco', 'ecod', 'code']
```

Either way, we're still operating on a specific variable here, `S` . To generalize a generator expression for an arbitrary subject, wrap it in a simple `lambda` function that takes an argument and returns a generator that uses it (subtle bit: note that `seq` in the second of these is not local to the generator, but is local to the `lambda` that encloses it):

```
>>> F = lambda seq: (seq[i:] + seq[:i] for i in range(l
>>> list(F(S))
['code', 'odec', 'deco', 'ecod']

>>> F = lambda seq: (seq:=(seq[1:] + seq[:1]) for i in
>>> list(F(S))
['odec', 'deco', 'ecod', 'code']

>>> list(F([1, 2, 3]))
[[2, 3, 1], [3, 1, 2], [1, 2, 3]]

>>> F(S)
<generator object <lambda>.<locals>.<genexpr> at 0x100b

>>> for x in F((1, 2, 3)): print(x, end=' ')
(2, 3, 1) (3, 1, 2) (1, 2, 3)
```

Tester client

Finally, we can use either the generator function or its expression equivalent in Chapter 18's `tester` to produce scrambled arguments. As packaged in the module of Example 20-1, the sequence scramblers become reusable tools.

Example 20-1. scramble.py

```
"Generate shuffles of a sequence, by function or expres

def scramble1(seq):
    for i in range(len(seq)):
        yield seq[i:] + seq[:i]        # Yield one shuf

scramble2 = lambda seq: (seq[i:] + seq[:i] for i in rar
```

Though it requires a bit of page flipping to see how, moving the values generation out to an external tool like this also makes the testing function noticeably simpler:

```
>>> from scramble import scramble1           # Choose y
>>> from inter2 import intersect, union      # Functior

>>> def tester(func, items, trace=True):
```

```
        for args in scramble1(items):         # Use eith
            if trace: print(args)
            print(sorted(func(*args)))        # Test for

>>> tester(intersect, ('aab', 'abcde', 'abababab'), False
['a', 'b']
['a', 'b']
['a', 'b']
```

To make this work for yourself, make sure all imported files are in the current directory: either copy *inter2.py* from Chapter 18's folder to Chapter 20's, or copy Example 20-1's *scramble.py* to Chapter 18's folder and work there (the examples package has already done the former). Alternatively, you can modify the module import search path's `PYTHONPATH` setting to include any folder—as you'll learn when we cover modules in full after the next chapter.

Permutating Sequences

Generators have many other real-world applications—consider parsing attachments in an email message or computing points to be plotted in a GUI. Moreover, other types of sequence scrambles serve central roles in other applications, from searches to mathematics. As is, our sequence scrambler of the prior section is a simple reordering, but some programs warrant the more exhaustive set of all possible orderings we get from *permutations*—produced using recursive functions in both list-builder and generator forms by the module file in Example 20-2.

Example 20-2. permute.py

```
"Permute sequences: as a list or generator of values"

def permute1(seq):
    if not seq:                                 # Shuffle
        return [seq]                            # Empty s
    else:
        res = []
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]          # Delete
            for x in permute1(rest):            # Permute
                res.append(seq[i:i+1] + x)      # Add noc
        return res
```

```
def permute2(seq):
    if not seq:                                 # Shuffle
        yield seq                               # Empty s
    else:
        for i in range(len(seq)):
            rest = seq[:i] + seq[i+1:]          # Delete
            for x in permute2(rest):            # Permute
                yield seq[i:i+1] + x            # Add noc
```

Both of these functions produce the same results, though the second defers much of its work until it is asked for a result. This code is a bit advanced, especially the second of these functions (and to some Python newcomers might even be categorized as cruel and unusual punishment!). Still, as you'll learn in a moment, there are cases where the generator approach can be very useful—and even essential.

Study and test this code for more insight, and add prints to trace if it helps. If it's still a mystery, try to make sense of the first version first; remember that generator functions simply return objects with methods that handle `next` operations run by `for` loops at each level, and don't produce any results until iterated; and trace through some of the following examples to see how they're handled by this code.

Permutations produce more orderings than the original scrambler—for `N` items, we get `N!` (factorial, not surprise) results instead of just `N` (24 for 4: 4 * 3 * 2 * 1). In fact, that's why we need *recursion* here: the number of nested loops is arbitrary, and depends on the length of the sequence permuted. Here's a demo of both shufflers at work:

```
>>> from scramble import scramble1
>>> from permute import permute1, permute2

>>> list(scramble1('abc'))                      # Si
['abc', 'bca', 'cab']

>>> permute1('abc')                             # Pe
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
>>> list(permute2('abc'))                       # Ge
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

>>> G = permute2('abc')                         # It
>>> next(G)
```

```
'abc'
>>> next(G)
'acb'
>>> for x in permute2('abc'): print(x)                # Au
…prints six lines…
```

The list and generator versions' results are the same, though the generator minimizes both space usage and delays for results. For larger items, the set of all permutations from both is much larger than the simpler scrambler's:

```
>>> permute1('hack') == list(permute2('hack'))
True
>>> len(list(permute2('hack'))), len(list(scramble1('ha
(24, 4)

>>> list(scramble1('hack'))
['hack', 'ackh', 'ckha', 'khac']
>>> list(permute2('hack'))
['hack', 'hakc', 'hcak', 'hcka', 'hkac', 'hkca', 'ahck'
 'akhc', 'akch', 'chak', 'chka', 'cahk', 'cakh', 'ckha'
 'kahc', 'kach', 'kcha', 'kcah']
```

Per Chapter 19, there are nonrecursive alternatives here too, using explicit stacks or queues, and other sequence orderings are common (e.g., fixed-size subsets and combinations that filter out duplicates of differing order), but these require coding extensions we'll forgo here. Experiment further on your own for more insights.

Why generators here: Space, time, and more

Generators are a somewhat advanced tool, and might be better treated as an optional topic, but for the fact that they permeate the Python language today. As we've seen, fundamental built-in tools such as `range`, `map`, dictionary `keys`, and even files are now generators, so you must be familiar with the concept even if you don't write new generators of your own. Moreover, user-defined generators are increasingly common in Python code that you might come across today—in the Python standard library, for instance.

Though your code is yours to code, the same guidelines given for list comprehensions apply here as well: don't complicate your code with user-defined generators if they are not warranted. Especially for smaller programs

and data sets, such tools may not make sense. Simple lists of results may suffice, may be easier to understand, will be garbage-collected automatically, and might be produced quicker (and are today: see the next chapter). Advanced tools like generators that rely on implicit "magic" can be fun to experiment with, but may be subpar when optional.

That being said, there are specific use cases that generators can address well. They can reduce memory footprint in some programs, reduce delays in others, and can occasionally make the impossible possible. Consider, for example, a program that must produce all possible permutations of a nontrivial sequence. Since the number of combinations is a *factorial* that explodes exponentially, the preceding `permute1` recursive list-builder function will either introduce a noticeable and perhaps interminable pause, or fail completely due to memory requirements.

By contrast, the `permute2` recursive generator returns each individual result quickly, and can handle very large result sets. Even at just 10 items, for instance, the difference starts becoming stark (per Python's `math` module):

```
>>> import math
>>> f'{math.factorial(10):,}'          # 10 * 9 * 8 * 7 *
'3,628,800'

>>> from permute import permute1, permute2
>>> seq = list(range(10))

>>> p1 = permute1(seq)                 # ~17 seconds on c
>>> len(p1)                            # Creates a list c
3628800
>>> p1[0], p1[-1]
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [9, 8, 7, 6, 5, 4, 3,
```

In this case, the `permute1` list builder pauses for 17 seconds to build a 3.6-million-item list, but the *generator* can begin returning individual results immediately—neither `permute2` nor any `next` pause in the following:

```
>>> p2 = permute2(seq)                 # Returns generato
>>> next(p2)                           # Returns each res
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> next(p2)
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
```

While collecting all results from the generator is still slow, it's faster than the list builder, and not intended usage:

```
>>> p2 = list(permute2(seq))          # About 10 seconds
>>> p1 == p2                          # Same set of resu
True
```

Naturally, we might be able to optimize the list builder's code to run quicker (e.g., an explicit stack instead of recursion might change its performance), but for larger sequences, it's not an option at all—at just 50 items, the number of permutations wholly precludes building a results list, and would take far too long for mere mortals like us in any event (and larger values will overflow the preset recursion stack depth limit: see the preceding chapter). The generator, however, is still viable—despite the factorial size of the problem, it is able to produce individual results immediately:

```
>>> math.factorial(50)
30414093201713378043612608166064768844377641568960512000

>>> p3 = permute2(list(range(50)))        # permute1 is
>>> next(p3)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
44, 45, 46, 47, 48, 49]
```

For more fun—and to yield results that are more variable and less obviously deterministic—we could use Python's `random` module of Chapter 5 to randomly shuffle the sequence to be permuted before the permuter begins its work. In the following, for example, each `permute2` and `next` call returns immediately as before and permutes a random sequence, but `permute1` hangs (and presumably perishes from memory starvation if allowed to run long enough):

```
>>> seq = list(range(20))
>>> print(f'{math.factorial(20):,}')      # permute1 is
2,432,902,008,176,640,000
```

```
>>> import random
>>> random.shuffle(seq)                    # Shuffle sequ
>>> p = permute2(seq)
>>> next(p)
[10, 19, 5, 6, 2, 13, 1, 8, 11, 7, 14, 16, 4, 3, 0, 18,
>>> next(p)
[10, 19, 5, 6, 2, 13, 1, 8, 11, 7, 14, 16, 4, 3, 0, 18,

>>> random.shuffle(seq)
>>> p = permute2(seq)
>>> next(p)
[17, 15, 14, 7, 10, 8, 2, 6, 18, 19, 13, 4, 1, 12, 5, 0
>>> next(p)
[17, 15, 14, 7, 10, 8, 2, 6, 18, 19, 13, 4, 1, 12, 5, 0
```

In fact, we might be able to use the random shuffler instead of manual shuffles in some roles, as long as we either can assume that it won't repeat shuffles during the time we consume them, or test its results against prior shuffles to avoid repeats—and hope that we do not live in the strange universe where a random sequence repeats the same result an infinite number of times! When full coverage is important, manual shuffles offer better control.

The main point here is that generators can sometimes produce results from large solution sets when list builders cannot. Then again, it's not clear how common such use cases may be in the real world, and this doesn't necessarily justify the *implicit* flavor of value generation that we get with generator functions and expressions. As you'll see in Part VI, value generation can also be coded as iterable objects with *classes*. Class-based iterables can produce items on request too, and are more *explicit* than the magic objects and methods produced for generator functions and expressions.

Part of programming is finding a balance among trade-offs like these, and there are no absolute rules here. While the benefits of generators may sometimes justify their use, maintainability counts too. Like comprehensions, generators also offer an *expressiveness* and *code economy* that's hard to resist if you understand how they work—but you'll want to weigh this against the frustration of coworkers who might not.

Example: Emulating zip and map

Let's take a quick look at one more example of generators in action that illustrates just how expressive they can be. Once you know about comprehensions, generators, and other iteration tools, you'll find that emulating many of Python's functional built-ins is both straightforward and instructive, and can be useful for custom roles.

For example, we've already seen how the built-in `zip` and `map` functions combine iterables and project functions across them, respectively. With multiple iterable arguments, `map` projects the function across items taken from each iterable in much the same way that `zip` combines them, and both functions truncate at the shortest iterable's length:

```
>>> list(zip('abc', 'xyz123'))                  # zip cc
[('a', 'x'), ('b', 'y'), ('c', 'z')]
>>> list(zip([1, 2, 3], [2, 3, 4, 5]))          # N M-ar
[(1, 2), (2, 3), (3, 4)]
>>> list(zip([-2, -1, 0, 1, 2]))                # 1 5-ar
[(-2,), (-1,), (0,), (1,), (2,)]

>>> list(map(abs, [-2, -1, 0, 1, 2]))           # Single
[2, 1, 0, 1, 2]
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5]))     # N iter
[1, 8, 81]
```

As covered earlier, both also work on any type of iterable, including files that read their lines automatically. Though they're ultimately used for different purposes, if you study these examples long enough, you might notice a relationship between `zip` results and `map` function arguments that our next example can exploit.

Coding Your Own map

Although the `map` and `zip` built-ins are fast and convenient, they're easy to implement in customizable code of our own. In the preceding chapter, for example, we wrote a function that emulated the `map` built-in for a *single* iterable argument. Per Example 20-3, it doesn't take much more work to allow for *multiple* iterables, as the built-in does.

Example 20-3. mymap-lists.py

```
"Emulate map: support multiple arguments, build a list

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [-2, -1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))
```

This version relies upon `*` argument syntax—it *collects* multiple sequence (really, iterable) arguments; *unpacks* them as `zip` arguments to combine; and then *unpacks* the combined `zip` results as arguments to the passed-in function. That is, we're using the fact that the zipping is essentially a nested operation in mapping. The test code at the bottom applies this to both one and two sequences to test—with the same results generated by the built-in `map`:

```
$ python3 mymap-list.py
[2, 1, 0, 1, 2]
[1, 8, 81]
```

Really, though, the preceding code exhibits the classic *list comprehension pattern*, building a list of operation results within a `for` loop. We can rewrite our mapper more concisely as an equivalent one-line list comprehension:

```
def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]
```

When this is run the result is the same as before, but the code is more concise and might run faster (more on performance in Chapter 21). Both of the preceding `mymap` versions build result lists all at once, though, and this can waste memory for larger lists. Now that we know about *generator* functions and expressions, it's simple to recode both these alternatives to produce results on demand instead, per Example 20-4.

Example 20-4. mymap-generate.py

```
"Emulate map: support multiple arguments, generate resu

def mymap_func(func, *seqs):
    for args in zip(*seqs):
        yield func(*args)

def mymap_expr(func, *seqs):
    return (func(*args) for args in zip(*seqs))

for mymap in (mymap_func, mymap_expr):
    print(list(mymap(abs, [-2, -1, 0, 1, 2])))
    print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))
```

Both these new versions produce the same results but return generators designed to support the iteration protocol—the first yields one result at a time explicitly, and the second returns a generator expression's result to do the same implicitly. As generators, we must wrap them in `list` calls to force them to produce their values all at once:

```
$ python3 mymap-generate.py
[2, 1, 0, 1, 2]
[1, 8, 81]
[2, 1, 0, 1, 2]
[1, 8, 81]
```

No real work is done here until the `list` calls force the generators to run by activating the iteration protocol. The `list` calls allow for display in testing, but this is not generally desirable in other contexts. The generators returned by these functions themselves, as well as the generators returned by the `zip` built-in they use, produce results only on demand, and that's the *whole point* of generators—delaying results saves memory space, and avoids pausing callers for full results.

Coding Your Own zip and 2.X map

Of course, much of the secret behind the success of the examples shown so far lies in their use of the `zip` built-in to combine arguments from multiple iterables. Using iteration tools, we can also code workalikes that emulate both

today's truncating `zip`, as well as the former padding behavior of Python 2.X's `map` when passed a `None` for its function—a still potentially useful tool despite its demise in 3.X, and the sort of thing that custom code can provide. Per Example 20-5, `zip` and this padding `map` are related in utility and nearly the same in code.

Example 20-5. myfptools-list.py

```
# Emulate zip and padding map: build lists

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res  = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res  = []
    while any(seqs):
        res.append(tuple((S.pop(0) if S else pad) for S
    return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))
```

Both of these functions work on any type of iterable object because they run their arguments through the `list` built-in to force result generation (e.g., files would work as arguments, in addition to sequences like strings). Notice the use of the `all` and `any` built-ins here—as we saw briefly in Chapter 14, these return `True` if all or any items in an iterable are `True` (or equivalently, nonempty), respectively. These built-ins are used to stop looping when any or all of the listified arguments become empty after deletions.

Also note the use of the *keyword-only* argument, `pad`; unlike the 2.X `map`, our version will allow any pad object to be specified. When these functions are run, the following results are printed—a `zip` and two padding `map`s:

```
$ python3 myfptools-list.py
[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2
```

These functions aren't amenable to list comprehension translation because their loops are too specific. As before, though, while our `zip` and `map` workalikes currently build and return result lists, it's just as easy to turn them into *generators* with `yield` so that they each return one piece of their result set at a time. Example 20-6 shows how.

Example 20-6. myfptools_generate.py

```
# Emulate zip and padding map: generate results

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in s
```

The results are the same as before, but this file assumes it will be used or tested elsewhere (it has no self-test code), and we need to use `list` again to force the generators to yield their values for display in the REPL:

```
$ python3
>>> from myfptools_generate import myzip, mymapPad
>>> list(myzip('abc', 'xyz123'))
[('a', 'x'), ('b', 'y'), ('c', 'z')]
>>> list(mymapPad('abc', 'xyz123', pad=99))
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2
```

Finally, here's an alternative implementation of our `zip` and `map` emulators—rather than deleting arguments from lists with the `pop` method, the following versions do their job by calculating the minimum and maximum

argument lengths. Armed with these lengths, it's easy to code nested list comprehensions to step through argument index ranges:

```
def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return [tuple(S[i] for S in seqs) for i in range(mi

def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index  = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S i
```

Because these use `len` and indexing, they assume that arguments are *sequences* or similar, not arbitrary iterables, much like our earlier sequence scramblers and permuters. The outer comprehensions here step through argument index ranges, and the inner comprehensions (passed to `tuple`) step through the passed-in sequences to pull out arguments in parallel (though not at the same time: see the next section!). When they're run, the results are as before.

Most strikingly, generators and iterators seem to run rampant in this example. The arguments passed to `min` and `max` are generator expressions, which run to completion before the nested comprehensions begin iterating. Moreover, the nested list comprehensions employ two levels of delayed evaluation—the `range` built-in is an iterable, as is the generator expression argument to `tuple`.

In fact, *no* results are produced here until the square brackets of the list comprehensions request values to place in the result list—which forces all the comprehensions and generators to run. To turn these functions *themselves* into generators instead of list builders, simply use parentheses instead of square brackets again. Here's the case for our zip:

```
def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(mi

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))         # Go!... [('a', 'x')
```

In this case, it takes a `list` call to activate the generators and other iterables to produce their results. Experiment with these on your own for more details. Developing further coding alternatives is left as a suggested exercise (but see the sidebar "Why You Will Care: Iteration Versus Python Morph" for an exploration of one such option). Here, we have time for just one more related tale from the generations saga.

Asynchronous Functions: The Short Story

Now that you've survived all the foregoing twists and turns of the functions story in Python, there's one last topic to go. Python 3.5 debuted an extension called *asynchronous* (usually shortened to *async*) functions, which are able to manually pause their execution while waiting for a result, in order to allow other such functions to run.

Such tools are available in other languages (e.g., JavaScript), which provided some of the inspiration and blueprint for their appearance in Python. Indeed, `match`, `:=`, f-strings, type hinting, and a host of other Python tools owe their presence to this same programming-languages arms race, whose logical outcome would make all languages the same.

Origins aside, the pathological use case for the async extension is input/output (IO) operations: a function can pause itself until an IO transfer completes so that other parts of the program can run during the wait. While other tools like multiple threads can and still do address this need too, some may judge async functions to be a lighter-weight option.

That said, there are downsides to this topic. For one thing, it has been morphing almost constantly since its first murmurs in Python 3.4, which makes it difficult to document in books with lifespans much longer than web pages and blogs. For another, it's something of an all-or-nothing proposition, because async code requires other async code.

More fundamentally, async functions are part of the applications-level domain of *parallel programming*—a category that also includes multiprocessing and multithreading. As such, they are a heavyweight subject of interest to only a subset of Python's user base, and a lot to ask of newcomers to Python or

programming in general. Especially given Python's already skyrocketing complexity, this may be best taken as an optional add-on topic, and out of scope here.

Nevertheless, async functions are not truly optional in Python: they were added deeply to the language as new syntax, via the `async` and `await` reserved words, and are now fair game in code you may have to reuse (and job-interview tests you may have to suffer). For all these reasons, this section provides just enough of an overview to whet your appetite, but defers to Python's online resources for more details when you're ready to tackle this advanced topic.

Async Basics

In a nutshell, async functions enable a *cooperative*, nonpreemptive multitasking model, with coding trade-offs similar to those imposed by asynchronous network servers. For instance, tasks must generally be short-lived to avoid monopolizing the host device's processor. Though begun earlier, Python 3.5 made this part of the language with the `async def` statement and `await` expression, as well as the `async for` iterator and `async with` context manager.

Async tools and syntax can be used only in functions defined with `async def`. Like the `yield` and `yield from` that this model extend, `async def` causes a function to be compiled specially: when called, instead of running its body, such a function returns an *awaitable* object that supports an expected method-based protocol.

To invoke this object, functions `await` results explicitly, or use an `async for` or `async with` that does so implicitly. Moreover, an *event loop* must be launched to run the show at large. Although the event loop runs just one task at a time (they're not truly parallel), it can switch to other tasks when a task runs an `await` for a pending result.

Running serial tasks with normal blocking calls

As usual, this may be easier to grok in code than narrative, so let's turn to some examples to demo the salient bits. All of this section's example snippets live in the file *all_async_demos.py* in the examples package, previewed in Example 20-7. To work along, ether cut and paste from this file or emedia, or

run the examples file in its entirety or with parts stubbed out with triple quotes. All examples here assume that the preamble at the top of Example 20-7 has been run.

Example 20-7. all_async_demos.py (preamble)

```
import time, asyncio
def now():
    return time.strftime('[%H:%M:%S]')      # Local tim

…all examples here…
```

We need a handful of standard-library tools here: some in the `asyncio` standard-library module are required, and we'll display times with `time.strftime` and pause with sleep calls in both `time` and `asyncio` to simulate a long-running task. See Python's library manual for these tools' fine print that we'll skip here for space and scope.

To get started, consider the following *non*-async code, in which a `main` function calls another, `producer`, that simply pauses for two seconds using the `time.sleep` call before returning a result. The sleep stands in for a real blocking operation—like an IO call, network transfer, or user interaction. As usual with normal functions, `main` simply waits for each `producer` call to run to completion and return a result, before it proceeds with a next step:

```
def producer(label):
    time.sleep(2)                                # Pause
    return f'All done, {label}, {now()}'         # And re

def main():
    print('Start =>', now())
    print(producer(f'serial task 1'))            # Run th
    print(producer(f'serial task 2'))            # Waitir
    print(producer(f'serial task 3'))            # Before
    print('Stop  =>', now())

main()
```

When run, this code produces the following mundane results. As you can see, each `producer` call runs a two-second sleep before another starts, so the

total time required is six seconds—two for each call run in series by `main`:

```
Start => [19:00:28]
All done, serial task 1, [19:00:30]
All done, serial task 2, [19:00:32]
All done, serial task 3, [19:00:34]
Stop  => [19:00:34]
```

Running concurrent tasks with "await" and "async def"

Now, let's do the same work with async functions. In the next listing, the `producer` is coded as an `async def` function, which makes Python compile it specially: it's now a *coroutine*—an object which, when called, doesn't produce its result, but instead returns an automatically created object that supports the coroutine method protocol. That protocol generally includes a method named `__await__`, which returns an iterator that produces coroutine results on `__next__`, but may instead use `__anext__` for generators. We can safely ignore all of these here.

The `async def` syntax is always required for functions that use `await`, as well as `aync for` and `async with` demoed ahead. Because `async def` is required to *both* use and define awaitable coroutines, it tends to spread viruslike throughout code (in fact, some tools now come with libraries in two separate forms, asynchronous and not).

Also in the following, `producer` uses `await` to suspend itself until the sleep expires. `await` is similar to a `yield from`, but the event loop is free to pause the coroutine running the `await` and resume another if the awaited result is not ready. Notice that `await` is used as a statement in `producer`; it's also an expression that returns the awaited result, as in `main`, where it returns the result of the awaited objects:

```
async def producer(label):                    # await
    await asyncio.sleep(2)                    # Call r
    return f'All done, {label}, {now()}'      # Result

async def main():
    print('Start =>', now())
    task1 = asyncio.create_task(producer(f'async task 1
    task2 = asyncio.create_task(producer(f'async task 2
    task3 = asyncio.create_task(producer(f'async task 3
```

```
    print(await task1)
    print(await task2)
    print(await task3)                          # Wait j
    print('Stop  =>', now())

asyncio.run(main())                             # Start
```

Crucially, the awaited `asyncio.sleep` call is like `time.sleep`, but is both *nonblocking*—it uses the event-loop's API to ensure that the caller doesn't pause for its completion—and *awaitable*—it uses `async def` to allow itself to be paused until a timeout elapses. Operations at the end of `await` chains must generally follow this model.

Also crucially, `main` uses `asyncio.create_task` to wrap each `producer` object in a *task*—an object run by the event loop that suspends execution of the coroutine it wraps while the coroutine waits for completion of a *future*, and returns the result of its wrapped coroutine when awaited itself. Future objects signal progress in ways that are too low level for this overview; here, it's enough to know that each task suspends itself while its coroutine is sleeping.

As a result, tasks can be overlapped in time, to run *concurrently*. In our code, all three tasks—and the `producer` coroutines they manage—finish at the same time, and the total program takes just two seconds instead of six:

```
Start => [19:00:34]
All done, async task 1, [19:00:36]
All done, async task 2, [19:00:36]
All done, async task 3, [19:00:36]
Stop  => [19:00:36]
```

Naturally, we can use loops here to make the steps more automated. The following runs the same as the prior version—taking two total seconds, and two for each overlapped task:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
```

```
    tasks = []
    for i in range(3):
        tasks.append(asyncio.create_task(producer(f'asy
    for task in tasks:
        print(await task)
    print('Stop  at', now())

asyncio.run(main())
```

In both `producer` and `main`, an `await`, much like a `yield from`, suspends execution of the coroutine that runs it until the awaited object returns its result. Together with the event loop and tasks, the net effect interleaves code.

How not to use async functions

To understand the preceding example, it may help to see other codings that do *not* work. In this example, the `await`s, tasks, and event loop are all essential: the code won't work without them. For one thing, calling `main` directly without routing it to the event loop fails right out of the gate with a warning. The call to `main` simply creates an awaitable object but does nothing with it (for brevity, snippets in this section show just differing code, followed by its results):

```
async def main():
    print(producer('xxx'))

main()

# Result…
…/all_async_blunders.py:12: RuntimeWarning: coroutine '
```

For another, calling `producer` directly instead of awaiting it fails too, and for the same reason. Much like generators, calling a coroutine returns the awaitable object, but doesn't run it; we must `await` the awaitable to make it go, just like we have to run `next` to get results from a generator:

```
async def main():
    print(producer('xxx'))

asyncio.run(main())
```

```
# Result…
<coroutine object producer at 0x10142e740>
…/all_async_blunders.py:22: RuntimeWarning: coroutine '
```

More vitally, tasks are key to the *concurrency* here. If the example's `main` simply runs an `await` for a `producer` directly, it will pause for the sleep too. In the following miscoding, both coroutines await the sleep's end, and the two `producer`s run serially—yielding a total of four seconds for the program at large instead of two:

```
async def main():
    print('Start =>', now())
    print(await producer('xxx'))
    print(await producer('yyy'))
    print('Stop  =>', now())

asyncio.run(main())

# Result…
Start => [18:41:56]
All done, xxx, [18:41:58]
All done, yyy, [18:42:00]
Stop  => [18:42:00]
```

This is also true if we defer the awaits by assignments— `main` continues to run after creating the `producer` coroutine awaitables, but still hangs for each later `await` that runs the coroutine's code, with no concurrency to be found:

```
async def main():
    print('Start =>', now())
    p1 = producer('xxx')
    p2 = producer('yyy')
    print('Await =>', now())
    print(await p1)
    print(await p2)
    print('Stop  =>', now())

asyncio.run(main())

# Result…
Start => [18:42:00]
```

```
Await => [18:42:00]
All done, xxx, [18:42:02]
All done, yyy, [18:42:04]
Stop  => [18:42:04]
```

That's why tasks are needed in the prior example: they enable coroutines to be paused and run by the event loop to overlap their execution. That is, they enable *concurrency*—along with alternatives like those of the next section.

Running concurrent tasks with "as_completed" and "gather"

But enough common mistakes: let's get back to coding concurrent coroutines the right way. Besides creating tasks, we can use the `gather` method to wait for all of our coroutines automatically, or `as_completed` to wait for results as they finish. The latter can be used in a normal `for` to watch for completions:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    coros = [producer(f'async task {i+1}') for i in rar
    for nextdone in asyncio.as_completed(coros):
        print(await nextdone)
    print('Stop  at', now())

asyncio.run(main())
```

This also finishes in two total seconds, though coroutine results may filter in in any order (`as_completed` can also be used with `async for` of the next section in ways covered by Python's manuals):

```
Start => [19:00:38]
All done, async task 2, [19:00:40]
All done, async task 3, [19:00:40]
All done, async task 1, [19:00:40]
Stop  at [19:00:40]
```

Alternatively, the `gather` call schedules tasks for coroutines, awaits all its arguments, and returns a list of task return values in the same order as its arguments when all have finished:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    coro1 = producer(f'async task 1')
    coro2 = producer(f'async task 2')
    coro3 = producer(f'async task 3')
    results = await asyncio.gather(coro1, coro2, coro3)
    print(results)
    print('Stop  at', now())

asyncio.run(main())
```

Results vary because they are in a list (formatted for display here), but it's still just two seconds for the entire gig:

```
Start => [19:00:40]
['All done, async task 1, [19:00:42]',
 'All done, async task 2, [19:00:42]',
 'All done, async task 3, [19:00:42]']
Stop  at [19:00:42]
```

For any line counters in the audience, a starred list comprehension can be used to create awaitables for `gather` more succinctly, and produces the same two-second result as the preceding version:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    print(await asyncio.gather(*[producer(f'async task
    print('Stop  at', now())
```

```
asyncio.run(main())
```

While this code works, Python's manuals today include a note that recommends `TaskGroup` over `gather` on the grounds of better exception handling. Should you care to follow that advice, you'll need to move on to the next section.

Running concurrent tasks with "async with" and "async for"

As yet another concurrency option, the combination of the `async with` statement and a manager object can automatically handle both protocols and exceptions, and wait for tasks to finish. As half of this pair, `async with` internally awaits the awaitables returned by its subject's `__aenter__` and `__aexit__` methods, the same way that the normal `with` statement calls the context-manager protocol's `__enter__` and `__exit__`.

Having said that, we unfortunately won't study either the normal `with` or its method protocols until Part VII, because they require knowledge of classes and OOP (yet again, Python's toolset assumes you must already know Python to use Python!), so you'll have to take this on faith for now. In the following code, though, it's enough to know that using a `TaskGroup` asynchronous context manager in concert with `async with` will automatically await results from all of its tasks before the statement exits, as well as handle some thorny issues involving exceptions:

```
async def producer(label):
    await asyncio.sleep(2)
    return f'All done, {label}, {now()}'

async def main():
    print('Start =>', now())
    async with asyncio.TaskGroup() as tg:
        tasks = [tg.create_task(producer(f'async task {
    for task in tasks:
        print(task.result())
    print('Stop  at', now())

asyncio.run(main())
```

The net effect runs all three `producer` calls concurrently in two seconds as before:

```
Start => [19:00:44]
All done, async task 1, [19:00:46]
All done, async task 2, [19:00:46]
All done, async task 3, [19:00:46]
Stop  at [19:00:46]
```

Finally, when a function is coded with `async def` and also uses `yield`, it's considered an *asynchronous generator*. When called, it returns an asynchronous iterable object which uses the `__anext__` method noted earlier, and can be used in an `async for` statement to execute the body of the function. (And no, there are no `async` variants of other statements like `while` or `if`; `with` and `for` were the only targets deemed worthy of the `async` twist—so far.)

Internally, `async for` uses methods `__aiter__` and `__anext__` much like the normal iteration protocol's `__iter__` and `__next__` that we studied in Chapter 14. The `__anext__` method returns an awaitable that produces a next value, and raises `StopAsyncIteration` to signal the end of values, instead of `StopIteration`. `async for` simply gets a next value by awaiting a call to `__anext__`, and running the loop's block of code for every value obtained this way. This encroaches on classes and OOP again, but statement users may ignore the details.

To avoid confusion, keep in mind that `async for` awaits the *next* item in the iterable it scans—which is not the same as awaiting for something else in the loop body of a normal `for`, as in prior examples here. Moreover, `async for` does not automatically *parallelize* anything—it's really just like a normal `for`, but adds an `await` for each next item obtained from its iterable, instead of issuing a potentially blocking call. `async for`'s implicit awaits between loops allow it to be interleaved with other ready-to-run tasks, but don't help much otherwise:

```
async def producer(label):
    for i in range(3):
        await asyncio.sleep(2)
        yield f'All done, {label} {i+1}, {now()}'
```

```
async def main():
    print('Start =>', now())
    async for reply in producer('async task'):
        print(reply)
    print('Stop  at', now())

asyncio.run(main())
```

As coded here, the result is the same six-second showing of the original serial version, because there's nothing else to run in this simple demo during the sleeps:

```
Start => [19:00:46]
All done, async task 1, [19:00:48]
All done, async task 2, [19:00:50]
All done, async task 3, [19:00:52]
Stop  at [19:00:52]
```

And lest this section hasn't yet managed to send you screaming into the night, it's also possible to use `await` and `async for` within a *comprehension* that's coded in an `async def` —but this must remain a story for another day:

```
async def hmm():
    result = [i async for i in asynciter() if i > 0]
    result = [await func() for func in coroutines]
```

The Async Wrap-Up

That's all the time and space we have for this topic (humanely, perhaps). Python's async functions may be more useful than type hinting (which, as we saw in Chapter 6, is both completely unused and at odds with Python's core ideas), but a full survey of either topic could fill chapters or books, and would be of interest to only a subset of Python users.

Moreover, this text isn't covering Python's multithreading or multiprocessing tools at all, even though they are at least as valid as async functions for parallelizing tasks. The fact that `async` alone was afforded dedicated syntax in the language both subjectively favors just one option in this domain and raises the bar for Python newcomers.

That said, your goals and views may differ, so please see Python's standard manuals for the full story if and when you wish to wade deeper into the async sea. Here, we must step back from the ledge of optional and stunningly convoluted extensions, and get back to the fundamentals that nearly all Python programs and programmers use.

Chapter Summary

This chapter wrapped up our coverage of built-in comprehension and iteration tools. It explored list comprehensions in the context of functional tools, and presented generator functions and expressions as additional iteration protocol tools. As a finale, we also explored some larger examples of iterations, comprehensions, and generations in action, and briefly glimpsed the asynchronous-functions extension to pique further study. Though we've now seen all the built-in iteration tools, the subject will resurface when we explore user-defined iterable classes in Chapter 30.

The next chapter is something of a continuation of the theme of this one—it rounds out this part of the book with a case study that times the performance of the tools we've studied here, and serves as a more realistic example at the midpoint of this book (yes, you're half done!). Before we move ahead to benchmarking comprehensions and generators, though, this chapter's quizzes give you a chance to review what you've learned about them here.

Test Your Knowledge: Quiz

1. What is the difference between enclosing a list comprehension in square brackets and parentheses?
2. How are generators and iterators related?
3. How can you tell if a function is a generator function?
4. What does a `yield` statement do?
5. How are `map` calls and list comprehensions related? Compare and contrast the two.
6. What do `async def` and `await` mean in a Python script?

Test Your Knowledge: Answers

1. List comprehensions in square brackets produce the result list all at once in memory. When they are enclosed in parentheses instead, they are actually generator expressions—they have the same internal syntax and a similar meaning but do not produce the result list all at once. Instead, generator expressions return a generator object, which yields one item in the result at a time when used in an iteration tool or iterated manually with `next`.
2. Generators are iterable objects that support the iteration protocol automatically—they have an iterator with a `__next__` method (run by `next`) that repeatedly advances to the next item in a series of results and raises an exception at the end of the series. In Python, we can code generator *functions* with `def` and `yield`, generator *expressions* with parenthesized comprehensions, and generator objects with *classes* that define a special method named `__iter__` (discussed later in the book).
3. A generator function has a `yield` statement (with or without its `from` extension) somewhere in its code. Generator functions are otherwise identical to normal functions syntactically, but they are compiled specially to return an iterable generator object when called. That object retains state and code location between values. (This means that deleting `yield` makes a function normal, but code deletions can cause all sorts of issues.)
4. When present, this statement makes Python compile the function specially as a generator; when called, the function returns a generator object that supports the iteration protocol. When the `yield` statement is run, it sends a result back to the caller and suspends the function's state; the function can then be resumed after the last `yield` statement, in response to a `next` built-in or `__next__` method call issued by the caller. In more advanced roles, the generator `send` method similarly resumes the generator, but can also pass a value that shows up as the `yield` expression's value. Generator functions may also have a `return` statement, which terminates the generator (and attaches an optional value to the automatic `StopIteration` exception).
5. The `map` call is similar to a list comprehension—both produce a series of values, by collecting the results of applying an operation to each item in a sequence or other iterable, one item at a time. The primary difference is that `map` applies a *function* call to each item, and list comprehensions apply arbitrary *expressions*. Because of this, list comprehensions are more general; they can apply a function call expression like `map`, but `map`

requires a function to apply other kinds of expressions. List comprehensions also support extended syntax—nested `for` loops, and `if` clauses that subsume the `filter` built-in. `map` also differs by producing a *generator* of values; the list comprehension materializes the result list in memory all at once, and this may matter for large lists.

6. The `async def` is used to define an asynchronous function (usually called a *coroutine*), and `await` waits for another asynchronous object to produce its value, possibly yielding control back to an event loop to allow other code to run during the wait. `async def` is required to use `await`, `async for`, and `async with`, and both an event loop and scheduling calls in module `asyncio` are generally required to achieve concurrency in this model. There's more to the async story than told here; see Python's manuals for next steps on this topic, as well as its multithreading and multiprocessing alternatives in Python's standard library.

WHY YOU WILL CARE: ITERATION VERSUS PYTHON MORPH

In Chapter 14, we noted that some built-ins (like `map`) support only a single traversal and are empty after it occurs, and this book promised to show you an example of why that can be important in practice. Now that we've studied iteration topics in full, it can make good on this promise—and demo the negative impacts of Python changes on your code at the same time.

In earlier editions of this book, the following clever coding for `zip` emulation, adapted from a version in Python's docs at the time, worked because `map` returned a physical list instead of a generator of values:

```
def myzip(*args):                                # Before 3.
    iters = map(iter, args)
    while iters:                                 # Guarantee
        res = [next(i) for i in iters]           # Any empty
        yield tuple(res)                         # Else retu
                                                 # Exit=retu
>>> list(myzip('ab', 'lmn'))
KeyboardInterrupt
```

Except this broke in Python 3.0. This code relied on that fact that `map`'s result supported multiple scans. When it mutated into a *single-scan* generator in 3.0, as soon as the list comprehension traversed `iters` once, the generator was exhausted but still `True`; later list comprehensions returned `[]`; and this code would loop until killed by a Ctrl+C. To fix, the prior edition added a `list` around `map` to make it reiterable:

```
def myzip(*args):                                # Before 3.
    iters = list(map(iter, args))                # <== Make
    while iters:                                 # map() is
        res = [next(i) for i in iters]
        yield tuple(res)

>>> list(myzip('ab', 'lmn'))
RuntimeError: generator raised StopIteration
```

Except this broke in Python 3.7. This code relied on the fact that an uncaught `StopIteration` from any iterable in `iters` was propagated through this function and sufficed to terminate its generation of values. Python 3.7 changed

to suppress this “bubbling” propagation of `StopIteration` through a generator and issue a `RunTimeError` in response. The workaround is to explicitly catch the `StopIteration` and `return` —even though `return` simply raises `StopIteration`:

```
def myzip(*args):                               # After 3.7
    iters = list(map(iter, args))
    while iters:
        try:                                    # <== Catch
            res = [next(i) for i in iters]      # StopIterat
        except StopIteration:
            return                              # How gener
        yield tuple(res)                        # But exits

>>> list(myzip('ab', 'lmn'))
[('a', 'l'), ('b', 'm')]
```

You’ll have to wait until Part VII for more on the `try` statement, and may have to similarly translate exceptions to returns this way in rare generators that rely on the former bubbling behavior. And while this final version works today, we’re not placing any bets on the future.

The takeaways here: wrapping `map` calls in `list` calls is not just for display; generators no longer propagate exit exceptions from code they run; and Python mods can and do break code—both subtly and repeatedly!