With a list comprehension, I can achieve the same outcome by specifying the expression for my computation along with the input sequence to loop over:

```
squares = [x**2 for x in a]  # List comprehension
print(squares)
```

```
>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unless you're applying a single-argument function, list comprehensions are also clearer than the `map` built-in function for simple cases. `map` requires the creation of a `lambda` function for the computation, which is visually noisy:

```
alt = map(lambda x: x ** 2, a)
```

Unlike `map`, list comprehensions let you easily filter items from the input `list`, removing corresponding outputs from the result. For example, say I want to compute the squares of the numbers that are divisible by 2. Here, I do this by adding a conditional expression to the list comprehension after the loop:

```
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)
```

```
>>>
[4, 16, 36, 64, 100]
```

The `filter` built-in function can be used along with `map` to achieve the same outcome, but it is much harder to read:

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

Dictionaries and sets have their own equivalents of list comprehensions (called *dictionary comprehensions* and *set comprehensions*, respectively). These make it easy to create other types of derivative data structures when writing algorithms:

```
even_squares_dict = {x: x**2 for x in a if x % 2 == 0}
threes_cubed_set = {x**3 for x in a if x % 3 == 0}
print(even_squares_dict)
print(threes_cubed_set)
```

```
>>>
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
{216, 729, 27}
```

Achieving the same outcome is possible with `map` and `filter` if you wrap each call with a corresponding constructor. These statements get so long that you have to break them up across multiple lines, which is even noisier and should be avoided:

```
alt_dict = dict(map(lambda x: (x, x**2),
                filter(lambda x: x % 2 == 0, a)))
alt_set = set(map(lambda x: x**3,
              filter(lambda x: x % 3 == 0, a)))
```

Things to Remember

- ✦ List comprehensions are clearer than the `map` and `filter` built-in functions because they don't require `lambda` expressions.
- ✦ List comprehensions allow you to easily skip items from the input `list`, a behavior that `map` doesn't support without help from `filter`.
- ✦ Dictionaries and sets may also be created using comprehensions.

Item 28: Avoid More Than Two Control Subexpressions in Comprehensions

Beyond basic usage (see Item 27: "Use Comprehensions Instead of `map` and `filter`"), comprehensions support multiple levels of looping. For example, say that I want to simplify a matrix (a `list` containing other `list` instances) into one flat `list` of all cells. Here, I do this with a list comprehension by including two `for` subexpressions. These subexpressions run in the order provided, from left to right:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
```

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

This example is simple, readable, and a reasonable usage of multiple loops in a comprehension. Another reasonable usage of multiple loops involves replicating the two-level-deep layout of the input `list`. For example, say that I want to square the value in each cell of a two-dimensional matrix. This comprehension is noisier because of the extra `[]` characters, but it's still relatively easy to read:

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)
```

```
>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

If this comprehension included another loop, it would get so long that I'd have to split it over multiple lines:

```
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    ...
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

At this point, the multiline comprehension isn't much shorter than the alternative. Here, I produce the same result using normal loop statements. The indentation of this version makes the looping clearer than the three-level-list comprehension:

```
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

Comprehensions support multiple `if` conditions. Multiple conditions at the same loop level have an implicit `and` expression. For example, say that I want to filter a `list` of numbers to only even values greater than 4. These two list comprehensions are equivalent:

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

Conditions can be specified at each level of looping after the `for` subexpression. For example, say I want to filter a matrix so the only cells remaining are those divisible by 3 in rows that sum to 10 or higher. Expressing this with a list comprehension does not require a lot of code, but it is extremely difficult to read:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)
```

```
>>>
[[6], [9]]
```

Although this example is a bit convoluted, in practice you'll see situations arise where such comprehensions seem like a good fit. I strongly encourage you to avoid using `list`, `dict`, or `set` comprehensions that look like this. The resulting code is very difficult for new readers to understand. The potential for confusion is even worse for dict comprehensions since they already need an extra parameter to represent both the key and the value for each item.

The rule of thumb is to avoid using more than two control subexpressions in a comprehension. This could be two conditions, two loops, or one condition and one loop. As soon as it gets more complicated than that, you should use normal if and for statements and write a helper function (see Item 30: "Consider Generators Instead of Returning Lists").

Things to Remember

- ✦ Comprehensions support multiple levels of loops and multiple conditions per loop level.
- ✦ Comprehensions with more than two control subexpressions are very difficult to read and should be avoided.

Item 29: Avoid Repeated Work in Comprehensions by Using Assignment Expressions

A common pattern with comprehensions—including list, dict, and set variants—is the need to reference the same computation in multiple places. For example, say that I'm writing a program to manage orders for a fastener company. As new orders come in from customers, I need to be able to tell them whether I can fulfill their orders. I need to verify that a request is sufficiently in stock and above the minimum threshold for shipping (in batches of 8):

```
stock = {
    'nails': 125,
    'screws': 35,
    'wingnuts': 8,
    'washers': 24,
}

order = ['screws', 'wingnuts', 'clips']

def get_batches(count, size):
    return count // size

result = {}
for name in order:
  count = stock.get(name, 0)
  batches = get_batches(count, 8)
  if batches:
    result[name] = batches

print(result)

>>>
{'screws': 4, 'wingnuts': 1}
```

Here, I implement this looping logic more succinctly using a dictionary comprehension (see Item 27: "Use Comprehensions Instead of map and filter" for best practices):

```
found = {name: get_batches(stock.get(name, 0), 8)
         for name in order
         if get_batches(stock.get(name, 0), 8)}
print(found)

>>>
{'screws': 4, 'wingnuts': 1}
```

Although this code is more compact, the problem with it is that the get_batches(stock.get(name, 0), 8) expression is repeated. This hurts readability by adding visual noise that's technically unnecessary. It also increases the likelihood of introducing a bug if the two expressions aren't kept in sync. For example, here I've changed the first get_batches call to have 4 as its second parameter instead of 8, which causes the results to be different:

```
has_bug = {name: get_batches(stock.get(name, 0), 4)
           for name in order
           if get_batches(stock.get(name, 0), 8)}

print('Expected:', found)
print('Found:   ', has_bug)

>>>
Expected: {'screws': 4, 'wingnuts': 1}
Found:    {'screws': 8, 'wingnuts': 2}
```

An easy solution to these problems is to use the walrus operator (:=), which was introduced in Python 3.8, to form an assignment expression as part of the comprehension (see Item 10: "Prevent Repetition with Assignment Expressions" for background):

```
found = {name: batches for name in order
         if (batches := get_batches(stock.get(name, 0), 8))}
```

The assignment expression (batches := get_batches(...)) allows me to look up the value for each order key in the stock dictionary a single time, call get_batches once, and then store its corresponding value in the batches variable. I can then reference that variable elsewhere in the comprehension to construct the dict's contents instead of having to call get_batches a second time. Eliminating the redundant calls to get and get_batches may also improve performance by avoiding unnecessary computations for each item in the order list.

It's valid syntax to define an assignment expression in the value expression for a comprehension. But if you try to reference the variable it defines in other parts of the comprehension, you might get an exception at runtime because of the order in which comprehensions are evaluated:

```
result = {name: (tenth := count // 10)
          for name, count in stock.items() if tenth > 0}
```

```
>>>
Traceback ...
NameError: name 'tenth' is not defined
```

I can fix this example by moving the assignment expression into the condition and then referencing the variable name it defined in the comprehension's value expression:

```
result = {name: tenth for name, count in stock.items()
          if (tenth := count // 10) > 0}
print(result)
```

```
>>>
{'nails': 12, 'screws': 3, 'washers': 2}
```

If a comprehension uses the walrus operator in the value part of the comprehension and doesn't have a condition, it'll leak the loop variable into the containing scope (see Item 21: "Know How Closures Interact with Variable Scope" for background):

```
half = [(last := count // 2) for count in stock.values()]
print(f'Last item of {half} is {last}')
```

```
>>>
Last item of [62, 17, 4, 12] is 12
```

This leakage of the loop variable is similar to what happens with a normal for loop:

```
for count in stock.values():  # Leaks loop variable
    pass
print(f'Last item of {list(stock.values())} is {count}')
```

```
>>>
Last item of [125, 35, 8, 24] is 24
```

However, similar leakage doesn’t happen for the loop variables from comprehensions:

```
half = [count // 2 for count in stock.values()]
print(half)   # Works
print(count)  # Exception because loop variable didn't leak
```

```
>>>
[62, 17, 4, 12]
Traceback ...
NameError: name 'count' is not defined
```

It’s better not to leak loop variables, so I recommend using assignment expressions only in the condition part of a comprehension.

Using an assignment expression also works the same way in generator expressions (see Item 32: “Consider Generator Expressions for Large List Comprehensions”). Here, I create an iterator of pairs containing the item name and the current count in stock instead of a `dict` instance:

```
found = ((name, batches) for name in order
         if (batches := get_batches(stock.get(name, 0), 8)))
print(next(found))
print(next(found))
```

```
>>>
('screws', 4)
('wingnuts', 1)
```

Things to Remember

- ✦ Assignment expressions make it possible for comprehensions and generator expressions to reuse the value from one condition elsewhere in the same comprehension, which can improve readability and performance.
- ✦ Although it’s possible to use an assignment expression outside of a comprehension or generator expression’s condition, you should avoid doing so.

Item 30: Consider Generators Instead of Returning Lists

The simplest choice for a function that produces a sequence of results is to return a `list` of items. For example, say that I want to find the index of every word in a string. Here, I accumulate results in a list using the append method and return it at the end of the function:

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

This works as expected for some sample input:

```
address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:10])
```

```
>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

There are two problems with the index_words function.

The first problem is that the code is a bit dense and noisy. Each time a new result is found, I call the append method. The method call's bulk (result.append) deemphasizes the value being added to the list (index + 1). There is one line for creating the result list and another for returning it. While the function body contains ~130 characters (without whitespace), only ~75 characters are important.

A better way to write this function is by using a *generator*. Generators are produced by functions that use yield expressions. Here, I define a generator function that produces the same results as before:

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```

When called, a generator function does not actually run but instead immediately returns an iterator. With each call to the next built-in function, the iterator advances the generator to its next yield expression. Each value passed to yield by the generator is returned by the iterator to the caller:

```
it = index_words_iter(address)
print(next(it))
print(next(it))
```

```
>>>
0
5
```

The index_words_iter function is significantly easier to read because all interactions with the result list have been eliminated. Results are passed to yield expressions instead. You can easily convert the iterator returned by the generator to a list by passing it to the list built-in function if necessary (see Item 32: "Consider Generator Expressions for Large List Comprehensions" for how this works):

```
result = list(index_words_iter(address))
print(result[:10])
```

```
>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

The second problem with index_words is that it requires all results to be stored in the list before being returned. For huge inputs, this can cause a program to run out of memory and crash.

In contrast, a generator version of this function can easily be adapted to take inputs of arbitrary length due to its bounded memory requirements. For example, here I define a generator that streams input from a file one line at a time and yields outputs one word at a time:

```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
        for letter in line:
            offset += 1
            if letter == ' ':
                yield offset
```

The working memory for this function is limited to the maximum length of one line of input. Running the generator produces the same results (see Item 36: "Consider itertools for Working with Iterators and Generators" for more about the islice function):

```
with open('address.txt', 'r') as f:
    it = index_file(f)
    results = itertools.islice(it, 0, 10)
    print(list(results))
```

```
>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

The only gotcha with defining generators like this is that the callers must be aware that the iterators returned are stateful and can't be reused (see Item 31: "Be Defensive When Iterating Over Arguments").

Things to Remember

- ✦ Using generators can be clearer than the alternative of having a function return a `list` of accumulated results.
- ✦ The iterator returned by a generator produces the set of values passed to `yield` expressions within the generator function's body.
- ✦ Generators can produce a sequence of outputs for arbitrarily large inputs because their working memory doesn't include all inputs and outputs.

Item 31: Be Defensive When Iterating Over Arguments

When a function takes a `list` of objects as a parameter, it's often important to iterate over that `list` multiple times. For example, say that I want to analyze tourism numbers for the U.S. state of Texas. Imagine that the data set is the number of visitors to each city (in millions per year). I'd like to figure out what percentage of overall tourism each city receives.

To do this, I need a normalization function that sums the inputs to determine the total number of tourists per year and then divides each city's individual visitor count by the total to find that city's contribution to the whole:

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

This function works as expected when given a `list` of visits:

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

To scale this up, I need to read the data from a file that contains every city in all of Texas. I define a generator to do this because then I can reuse the same function later, when I want to compute tourism numbers for the whole world—a much larger data set with higher memory requirements (see Item 30: "Consider Generators Instead of Returning Lists" for background):

```
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)
```

Surprisingly, calling `normalize` on the `read_visits` generator's return value produces no results:

```
it = read_visits('my_numbers.txt')
percentages = normalize(it)
print(percentages)
```

```
>>>
[]
```

This behavior occurs because an iterator produces its results only a single time. If you iterate over an iterator or a generator that has already raised a `StopIteration` exception, you won't get any results the second time around:

```
it = read_visits('my_numbers.txt')
print(list(it))
print(list(it))  # Already exhausted
```

```
>>>
[15, 35, 80]
[]
```

Confusingly, you also won't get errors when you iterate over an already exhausted iterator. `for` loops, the `list` constructor, and many other functions throughout the Python standard library expect the `StopIteration` exception to be raised during normal operation. These functions can't tell the difference between an iterator that has no output and an iterator that had output and is now exhausted.

To solve this problem, you can explicitly exhaust an input iterator and keep a copy of its entire contents in a `list`. You can then iterate over the `list` version of the data as many times as you need to. Here's the same function as before, but it defensively copies the input iterator:

```
def normalize_copy(numbers):
    numbers_copy = list(numbers)  # Copy the iterator
    total = sum(numbers_copy)
    result = []
    for value in numbers_copy:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Now the function works correctly on the `read_visits` generator's return value:

```
it = read_visits('my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

The problem with this approach is that the copy of the input iterator's contents could be extremely large. Copying the iterator could cause the program to run out of memory and crash. This potential for scalability issues undermines the reason that I wrote `read_visits` as a generator in the first place. One way around this is to accept a function that returns a new iterator each time it's called:

```
def normalize_func(get_iter):
    total = sum(get_iter())   # New iterator
    result = []
    for value in get_iter():  # New iterator
        percent = 100 * value / total
        result.append(percent)
    return result
```

To use `normalize_func`, I can pass in a `lambda` expression that calls the generator and produces a new iterator each time:

```
path = 'my_numbers.txt'
percentages = normalize_func(lambda: read_visits(path))
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

Although this works, having to pass a lambda function like this is clumsy. A better way to achieve the same result is to provide a new container class that implements the *iterator protocol*.

The iterator protocol is how Python for loops and related expressions traverse the contents of a container type. When Python sees a statement like for x in foo, it actually calls iter(foo). The iter built-in function calls the foo.__iter__ special method in turn. The __iter__ method must return an iterator object (which itself implements the __next__ special method). Then, the for loop repeatedly calls the next built-in function on the iterator object until it's exhausted (indicated by raising a StopIteration exception).

It sounds complicated, but practically speaking, you can achieve all of this behavior for your classes by implementing the __iter__ method as a generator. Here, I define an iterable container class that reads the file containing tourism data:

```
class ReadVisits:
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
        with open(self.data_path) as f:
            for line in f:
                yield int(line)
```

This new container type works correctly when passed to the original function without modifications:

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

This works because the sum method in normalize calls ReadVisits.__iter__ to allocate a new iterator object. The for loop to normalize the numbers also calls __iter__ to allocate a second iterator object. Each of those iterators will be advanced and exhausted independently, ensuring that each unique iteration sees all of the input data values. The only downside of this approach is that it reads the input data multiple times.

Now that you know how containers like ReadVisits work, you can write your functions and methods to ensure that parameters aren't just iterators. The protocol states that when an iterator is passed to the iter built-in function, iter returns the iterator itself. In contrast, when a container type is passed to iter, a new iterator object is returned each time. Thus, you can test an input value for this behavior and raise a TypeError to reject arguments that can't be repeatedly iterated over:

```
def normalize_defensive(numbers):
    if iter(numbers) is numbers:  # An iterator -- bad!
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Alternatively, the collections.abc built-in module defines an Iterator class that can be used in an isinstance test to recognize the potential problem (see Item 43: "Inherit from collections.abc for Custom Container Types"):

```
from collections.abc import Iterator


def normalize_defensive(numbers):
    if isinstance(numbers, Iterator):  # Another way to check
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

The approach of using a container is ideal if you don't want to copy the full input iterator, as with the normalize_copy function above, but you also need to iterate over the input data multiple times. This function works as expected for list and ReadVisits inputs because they are iterable containers that follow the iterator protocol:

```
visits = [15, 35, 80]
percentages = normalize_defensive(visits)
assert sum(percentages) == 100.0

visits = ReadVisits(path)
percentages = normalize_defensive(visits)
assert sum(percentages) == 100.0
```

The function raises an exception if the input is an iterator rather than a container:

```
visits = [15, 35, 80]
it = iter(visits)
normalize_defensive(it)
```

```
>>>
Traceback ...
TypeError: Must supply a container
```

The same approach can also be used for asynchronous iterators (see Item 61: "Know How to Port Threaded I/O to `asyncio`" for an example).

Things to Remember

- ✦ Beware of functions and methods that iterate over input arguments multiple times. If these arguments are iterators, you may see strange behavior and missing values.
- ✦ Python's iterator protocol defines how containers and iterators interact with the `iter` and `next` built-in functions, `for` loops, and related expressions.
- ✦ You can easily define your own iterable container type by implementing the `__iter__` method as a generator.
- ✦ You can detect that a value is an iterator (instead of a container) if calling `iter` on it produces the same value as what you passed in. Alternatively, you can use the `isinstance` built-in function along with the `collections.abc.Iterator` class.

Item 32: Consider Generator Expressions for Large List Comprehensions

The problem with list comprehensions (see Item 27: "Use Comprehensions Instead of `map` and `filter`") is that they may create new `list` instances containing one item for each value in input sequences. This is fine for small inputs, but for large inputs, this behavior could consume significant amounts of memory and cause a program to crash.

For example, say that I want to read a file and return the number of characters on each line. Doing this with a list comprehension would require holding the length of every line of the file in memory. If the file is enormous or perhaps a never-ending network socket, using list comprehensions would be problematic. Here, I use a list comprehension in a way that can only handle small input values:

```
value = [len(x) for x in open('my_file.txt')]
print(value)
```

```
>>>
[100, 57, 15, 1, 12, 75, 5, 86, 89, 11]
```

To solve this issue, Python provides *generator expressions*, which are a generalization of list comprehensions and generators. Generator expressions don't materialize the whole output sequence when they're run. Instead, generator expressions evaluate to an iterator that yields one item at a time from the expression.

You create a generator expression by putting list-comprehension-like syntax between () characters. Here, I use a generator expression that is equivalent to the code above. However, the generator expression immediately evaluates to an iterator and doesn't make forward progress:

```
it = (len(x) for x in open('my_file.txt'))
print(it)
```

```
>>>
<generator object <genexpr> at 0x108993dd0>
```

The returned iterator can be advanced one step at a time to produce the next output from the generator expression, as needed (using the `next` built-in function). I can consume as much of the generator expression as I want without risking a blowup in memory usage:

```
print(next(it))
print(next(it))
```

```
>>>
100
57
```

Another powerful outcome of generator expressions is that they can be composed together. Here, I take the iterator returned by the generator expression above and use it as the input for another generator expression:

```
roots = ((x, x**0.5) for x in it)
```

Each time I advance this iterator, it also advances the interior iterator, creating a domino effect of looping, evaluating conditional expressions, and passing around inputs and outputs, all while being as memory efficient as possible:

```
print(next(roots))
```

```
>>>
(15, 3.872983346207417)
```

Chaining generators together like this executes very quickly in Python. When you're looking for a way to compose functionality that's operating on a large stream of input, generator expressions are a great choice. The only gotcha is that the iterators returned by generator expressions are stateful, so you must be careful not to use these iterators more than once (see Item 31: "Be Defensive When Iterating Over Arguments").

Things to Remember

- ✦ List comprehensions can cause problems for large inputs by using too much memory.
- ✦ Generator expressions avoid memory issues by producing outputs one at a time as iterators.
- ✦ Generator expressions can be composed by passing the iterator from one generator expression into the `for` subexpression of another.
- ✦ Generator expressions execute very quickly when chained together and are memory efficient.

Item 33: Compose Multiple Generators with `yield from`

Generators provide a variety of benefits (see Item 30: "Consider Generators Instead of Returning Lists") and solutions to common problems (see Item 31: "Be Defensive When Iterating Over Arguments"). Generators are so useful that many programs start to look like layers of generators strung together.

For example, say that I have a graphical program that's using generators to animate the movement of images onscreen. To get the visual effect I'm looking for, I need the images to move quickly at first, pause temporarily, and then continue moving at a slower pace. Here, I define two generators that yield the expected onscreen deltas for each part of this animation:

```
def move(period, speed):
    for _ in range(period):
        yield speed

def pause(delay):
    for _ in range(delay):
        yield 0
```

To create the final animation, I need to combine `move` and `pause` together to produce a single sequence of onscreen deltas. Here, I do this by calling a generator for each step of the animation, iterating over each generator in turn, and then yielding the deltas from all of them in sequence:

```
def animate():
    for delta in move(4, 5.0):
        yield delta
    for delta in pause(3):
        yield delta
    for delta in move(2, 3.0):
        yield delta
```

Now, I can render those deltas onscreen as they're produced by the single `animation` generator:

```
def render(delta):
    print(f'Delta: {delta:.1f}')
    # Move the images onscreen
    ...

def run(func):
    for delta in func():
        render(delta)

run(animate)
```

```
>>>
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 0.0
Delta: 0.0
Delta: 0.0
Delta: 3.0
Delta: 3.0
```

The problem with this code is the repetitive nature of the `animate` function. The redundancy of the `for` statements and `yield` expressions for each generator adds noise and reduces readability. This example includes only three nested generators and it's already hurting clarity; a complex animation with a dozen phases or more would be extremely difficult to follow.

The solution to this problem is to use the `yield from` expression. This advanced generator feature allows you to yield all values from a nested generator before returning control to the parent generator. Here, I reimplement the `animation` function by using `yield from`:

```
def animate_composed():
    yield from move(4, 5.0)
    yield from pause(3)
    yield from move(2, 3.0)

run(animate_composed)
```

```
>>>
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 0.0
Delta: 0.0
Delta: 0.0
Delta: 3.0
Delta: 3.0
```

The result is the same as before, but now the code is clearer and more intuitive. `yield from` essentially causes the Python interpreter to handle the nested `for` loop and `yield` expression boilerplate for you, which results in better performance. Here, I verify the speedup by using the `timeit` built-in module to run a micro-benchmark:

```
import timeit


def child():
    for i in range(1_000_000):
        yield i

def slow():
    for i in child():
        yield i

def fast():
    yield from child()

baseline = timeit.timeit(
    stmt='for _ in slow(): pass',
    globals=globals(),
    number=50)
print(f'Manual nesting {baseline:.2f}s')
```

```
comparison = timeit.timeit(
    stmt='for _ in fast(): pass',
    globals=globals(),
    number=50)
print(f'Composed nesting {comparison:.2f}s')

reduction = -(comparison - baseline) / baseline
print(f'{reduction:.1%} less time')
```

```
>>>
Manual nesting 4.02s
Composed nesting 3.47s
13.5% less time
```

If you find yourself composing generators, I strongly encourage you to use `yield from` when possible.

Things to Remember

- ✦ The `yield from` expression allows you to compose multiple nested generators together into a single combined generator.
- ✦ `yield from` provides better performance than manually iterating nested generators and yielding their outputs.

Item 34: Avoid Injecting Data into Generators with `send`

`yield` expressions provide generator functions with a simple way to produce an iterable series of output values (see Item 30: "Consider Generators Instead of Returning Lists"). However, this channel appears to be unidirectional: There's no immediately obvious way to simultaneously stream data in and out of a generator as it runs. Having such bidirectional communication could be valuable for a variety of use cases.

For example, say that I'm writing a program to transmit signals using a software-defined radio. Here, I use a function to generate an approximation of a sine wave with a given number of points:

```
import math


def wave(amplitude, steps):
    step_size = 2 * math.pi / steps
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        output = amplitude * fraction
        yield output
```

Now, I can transmit the wave signal at a single specified amplitude by iterating over the wave generator:

```
def transmit(output):
    if output is None:
        print(f'Output is None')
    else:
        print(f'Output: {output:>5.1f}')

def run(it):
    for output in it:
        transmit(output)

run(wave(3.0, 8))

>>>
Output:   0.0
Output:   2.1
Output:   3.0
Output:   2.1
Output:   0.0
Output:  -2.1
Output:  -3.0
Output:  -2.1
```

This works fine for producing basic waveforms, but it can't be used to constantly vary the amplitude of the wave based on a separate input (i.e., as required to broadcast AM radio signals). I need a way to modulate the amplitude on each iteration of the generator.

Python generators support the `send` method, which upgrades `yield` expressions into a two-way channel. The `send` method can be used to provide streaming inputs to a generator at the same time it's yielding outputs. Normally, when iterating a generator, the value of the `yield` expression is `None`:

```
def my_generator():
    received = yield 1
    print(f'received = {received}')

it = iter(my_generator())
output = next(it)       # Get first generator output
print(f'output = {output}')
```

```
try:
    next(it)             # Run generator until it exits
except StopIteration:
    pass

>>>
output = 1
received = None
```

When I call the send method instead of iterating the generator with a for loop or the next built-in function, the supplied parameter becomes the value of the yield expression when the generator is resumed. However, when the generator first starts, a yield expression has not been encountered yet, so the only valid value for calling send initially is None (any other argument would raise an exception at runtime):

```
it = iter(my_generator())
output = it.send(None)  # Get first generator output
print(f'output = {output}')

try:
    it.send('hello!')   # Send value into the generator
except StopIteration:
    pass

>>>
output = 1
received = hello!
```

I can take advantage of this behavior in order to modulate the amplitude of the sine wave based on an input signal. First, I need to change the wave generator to save the amplitude returned by the yield expression and use it to calculate the next generated output:

```
def wave_modulating(steps):
    step_size = 2 * math.pi / steps
    amplitude = yield             # Receive initial amplitude
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        output = amplitude * fraction
        amplitude = yield output  # Receive next amplitude
```

Then, I need to update the run function to stream the modulating amplitude into the wave_modulating generator on each iteration. The first input to send must be None, since a yield expression would not have occurred within the generator yet:

```
def run_modulating(it):
    amplitudes = [
        None, 7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]
    for amplitude in amplitudes:
        output = it.send(amplitude)
        transmit(output)

run_modulating(wave_modulating(12))
```

```
>>>
Output is None
Output:   0.0
Output:   3.5
Output:   6.1
Output:   2.0
Output:   1.7
Output:   1.0
Output:   0.0
Output:  -5.0
Output:  -8.7
Output: -10.0
Output:  -8.7
Output:  -5.0
```

This works; it properly varies the output amplitude based on the input signal. The first output is None, as expected, because a value for the amplitude wasn't received by the generator until after the initial yield expression.

One problem with this code is that it's difficult for new readers to understand: Using yield on the right side of an assignment statement isn't intuitive, and it's hard to see the connection between yield and send without already knowing the details of this advanced generator feature.

Now, imagine that the program's requirements get more complicated. Instead of using a simple sine wave as my carrier, I need to use a complex waveform consisting of multiple signals in sequence. One way to implement this behavior is by composing multiple generators together by using the yield from expression (see Item 33: "Compose Multiple Generators with yield from"). Here, I confirm that this works as expected in the simpler case where the amplitude is fixed:

```
def complex_wave():
    yield from wave(7.0, 3)
    yield from wave(2.0, 4)
    yield from wave(10.0, 5)

run(complex_wave())

>>>
Output:   0.0
Output:   6.1
Output:  -6.1
Output:   0.0
Output:   2.0
Output:   0.0
Output:  -2.0
Output:   0.0
Output:   9.5
Output:   5.9
Output:  -5.9
Output:  -9.5
```

Given that the yield from expression handles the simpler case, you may expect it to also work properly along with the generator send method. Here, I try to use it this way by composing multiple calls to the wave_modulating generator together:

```
def complex_wave_modulating():
    yield from wave_modulating(3)
    yield from wave_modulating(4)
    yield from wave_modulating(5)

run_modulating(complex_wave_modulating())

>>>
Output is None
Output:   0.0
Output:   6.1
Output:  -6.1
Output is None
Output:   0.0
Output:   2.0
Output:   0.0
Output: -10.0
Output is None
Output:   0.0
Output:   9.5
Output:   5.9
```

This works to some extent, but the result contains a big surprise: There are many None values in the output! Why does this happen? When each yield from expression finishes iterating over a nested generator, it moves on to the next one. Each nested generator starts with a bare yield expression—one without a value—in order to receive the initial amplitude from a generator send method call. This causes the parent generator to output a None value when it transitions between child generators.

This means that assumptions about how the yield from and send features behave individually will be broken if you try to use them together. Although it's possible to work around this None problem by increasing the complexity of the run_modulating function, it's not worth the trouble. It's already difficult for new readers of the code to understand how send works. This surprising gotcha with yield from makes it even worse. My advice is to avoid the send method entirely and go with a simpler approach.

The easiest solution is to pass an iterator into the wave function. The iterator should return an input amplitude each time the next built-in function is called on it. This arrangement ensures that each generator is progressed in a cascade as inputs and outputs are processed (see Item 32: "Consider Generator Expressions for Large List Comprehensions" for another example):

```
def wave_cascading(amplitude_it, steps):
    step_size = 2 * math.pi / steps
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        amplitude = next(amplitude_it)  # Get next input
        output = amplitude * fraction
        yield output
```

I can pass the same iterator into each of the generator functions that I'm trying to compose together. Iterators are stateful (see Item 31: "Be Defensive When Iterating Over Arguments"), and thus each of the nested generators picks up where the previous generator left off:

```
def complex_wave_cascading(amplitude_it):
    yield from wave_cascading(amplitude_it, 3)
    yield from wave_cascading(amplitude_it, 4)
    yield from wave_cascading(amplitude_it, 5)
```

Now, I can run the composed generator by simply passing in an iterator from the amplitudes list:

```
def run_cascading():
    amplitudes = [7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]
    it = complex_wave_cascading(iter(amplitudes))
    for amplitude in amplitudes:
        output = next(it)
        transmit(output)

run_cascading()
```

```
>>>
Output:   0.0
Output:   6.1
Output:  -6.1
Output:   0.0
Output:   2.0
Output:   0.0
Output:  -2.0
Output:   0.0
Output:   9.5
Output:   5.9
Output:  -5.9
Output:  -9.5
```

The best part about this approach is that the iterator can come from anywhere and could be completely dynamic (e.g., implemented using a generator function). The only downside is that this code assumes that the input generator is completely thread safe, which may not be the case. If you need to cross thread boundaries, async functions may be a better fit (see Item 62: "Mix Threads and Coroutines to Ease the Transition to asyncio").

Things to Remember

- ✦ The send method can be used to inject data into a generator by giving the yield expression a value that can be assigned to a variable.
- ✦ Using send with yield from expressions may cause surprising behavior, such as None values appearing at unexpected times in the generator output.
- ✦ Providing an input iterator to a set of composed generators is a better approach than using the send method, which should be avoided.

Item 35: Avoid Causing State Transitions in Generators with throw

In addition to yield from expressions (see Item 33: "Compose Multiple Generators with yield from") and the send method (see Item 34: "Avoid Injecting Data into Generators with send"), another advanced generator feature is the `throw` method for re-raising `Exception` instances within generator functions. The way `throw` works is simple: When the method is called, the next occurrence of a `yield` expression re-raises the provided `Exception` instance after its output is received instead of continuing normally. Here, I show a simple example of this behavior in action:

```
class MyError(Exception):
    pass

def my_generator():
    yield 1
    yield 2
    yield 3

it = my_generator()
print(next(it))  # Yield 1
print(next(it))  # Yield 2
print(it.throw(MyError('test error')))
```

```
>>>
1
2
Traceback ...
MyError: test error
```

When you call `throw`, the generator function may catch the injected exception with a standard `try/except` compound statement that surrounds the last `yield` expression that was executed (see Item 65: "Take Advantage of Each Block in `try/except/else/finally`" for more about exception handling):

```
def my_generator():
    yield 1

    try:
        yield 2
    except MyError:
        print('Got MyError!')
    else:
        yield 3

    yield 4

it = my_generator()
print(next(it))  # Yield 1
print(next(it))  # Yield 2
print(it.throw(MyError('test error')))
```

```
>>>
1
2
Got MyError!
4
```

This functionality provides a two-way communication channel between a generator and its caller that can be useful in certain situations (see Item 34: "Avoid Injecting Data into Generators with send" for another one). For example, imagine that I'm trying to write a program with a timer that supports sporadic resets. Here, I implement this behavior by defining a generator that relies on the throw method:

```
class Reset(Exception):
    pass

def timer(period):
    current = period
    while current:
        current -= 1
        try:
            yield current
        except Reset:
            current = period
```

In this code, whenever the Reset exception is raised by the yield expression, the counter resets itself to its original period.

I can connect this counter reset event to an external input that's polled every second. Then, I can define a run function to drive the timer generator, which injects exceptions with throw to cause resets, or calls announce for each generator output:

```
def check_for_reset():
    # Poll for external event
    ...

def announce(remaining):
    print(f'{remaining} ticks remaining')

def run():
    it = timer(4)
    while True:
        try:
            if check_for_reset():
                current = it.throw(Reset())
            else:
                current = next(it)
        except StopIteration:
            break
        else:
            announce(current)

run()

>>>
3 ticks remaining
2 ticks remaining
1 ticks remaining
3 ticks remaining
2 ticks remaining
3 ticks remaining
2 ticks remaining
1 ticks remaining
0 ticks remaining
```

This code works as expected, but it's much harder to read than necessary. The various levels of nesting required to catch `StopIteration` exceptions or decide to `throw`, call `next`, or `announce` make the code noisy.

A simpler approach to implementing this functionality is to define a stateful closure (see Item 38: "Accept Functions Instead of Classes for Simple Interfaces") using an iterable container object (see Item 31: "Be Defensive When Iterating Over Arguments"). Here, I redefine the `timer` generator by using such a class:

```
class Timer:
    def __init__(self, period):
        self.current = period
        self.period = period

    def reset(self):
        self.current = self.period

    def __iter__(self):
        while self.current:
            self.current -= 1
            yield self.current
```

Now, the run method can do a much simpler iteration by using a for statement, and the code is much easier to follow because of the reduction in the levels of nesting:

```
def run():
    timer = Timer(4)
    for current in timer:
        if check_for_reset():
            timer.reset()
        announce(current)

run()
```

```
>>>
3 ticks remaining
2 ticks remaining
1 ticks remaining
3 ticks remaining
2 ticks remaining
3 ticks remaining
2 ticks remaining
1 ticks remaining
0 ticks remaining
```

The output matches the earlier version using throw, but this implementation is much easier to understand, especially for new readers of the code. Often, what you're trying to accomplish by mixing generators and exceptions is better achieved with asynchronous features (see Item 60: "Achieve Highly Concurrent I/O with Coroutines"). Thus, I suggest that you avoid using throw entirely and instead use an iterable class if you need this type of exceptional behavior.

Things to Remember

✦ The throw method can be used to re-raise exceptions within generators at the position of the most recently executed yield expression.

✦ Using throw harms readability because it requires additional nesting and boilerplate in order to raise and catch exceptions.

✦ A better way to provide exceptional behavior in generators is to use a class that implements the __iter__ method along with methods to cause exceptional state transitions.

Item 36: Consider `itertools` for Working with Iterators and Generators

The `itertools` built-in module contains a large number of functions that are useful for organizing and interacting with iterators (see Item 30: "Consider Generators Instead of Returning Lists" and Item 31: "Be Defensive When Iterating Over Arguments" for background):

```
import itertools
```

Whenever you find yourself dealing with tricky iteration code, it's worth looking at the `itertools` documentation again to see if there's anything in there for you to use (see `help(itertools)`). The following sections describe the most important functions that you should know in three primary categories.

Linking Iterators Together

The `itertools` built-in module includes a number of functions for linking iterators together.

chain

Use `chain` to combine multiple iterators into a single sequential iterator:

```
it = itertools.chain([1, 2, 3], [4, 5, 6])
print(list(it))
```

```
>>>
[1, 2, 3, 4, 5, 6]
```

repeat

Use `repeat` to output a single value forever, or use the second parameter to specify a maximum number of times:

```
it = itertools.repeat('hello', 3)
print(list(it))
```

```
>>>
['hello', 'hello', 'hello']
```

cycle

Use `cycle` to repeat an iterator's items forever:

```
it = itertools.cycle([1, 2])
result = [next(it) for _ in range (10)]
print(result)
```

```
>>>
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

tee

Use tee to split a single iterator into the number of parallel iterators specified by the second parameter. The memory usage of this function will grow if the iterators don't progress at the same speed since buffering will be required to enqueue the pending items:

```
it1, it2, it3 = itertools.tee(['first', 'second'], 3)
print(list(it1))
print(list(it2))
print(list(it3))
```

```
>>>
['first', 'second']
['first', 'second']
['first', 'second']
```

zip_longest

This variant of the zip built-in function (see Item 8: "Use zip to Process Iterators in Parallel") returns a placeholder value when an iterator is exhausted, which may happen if iterators have different lengths:

```
keys = ['one', 'two', 'three']
values = [1, 2]

normal = list(zip(keys, values))
print('zip:         ', normal)

it = itertools.zip_longest(keys, values, fillvalue='nope')
longest = list(it)
print('zip_longest:', longest)
```

```
>>>
zip:         [('one', 1), ('two', 2)]
zip_longest: [('one', 1), ('two', 2), ('three', 'nope')]
```

Filtering Items from an Iterator

The itertools built-in module includes a number of functions for filtering items from an iterator.

islice

Use islice to slice an iterator by numerical indexes without copying. You can specify the end, start and end, or start, end, and step sizes, and the behavior is similar to that of standard sequence slicing and striding (see Item 11: "Know How to Slice Sequences" and Item 12: "Avoid Striding and Slicing in a Single Expression"):

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

first_five = itertools.islice(values, 5)
print('First five: ', list(first_five))

middle_odds = itertools.islice(values, 2, 8, 2)
print('Middle odds:', list(middle_odds))
```

```
>>>
First five:  [1, 2, 3, 4, 5]
Middle odds: [3, 5, 7]
```

takewhile

takewhile returns items from an iterator until a predicate function returns False for an item:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.takewhile(less_than_seven, values)
print(list(it))
```

```
>>>
[1, 2, 3, 4, 5, 6]
```

dropwhile

dropwhile, which is the opposite of takewhile, skips items from an iterator until the predicate function returns True for the first time:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.dropwhile(less_than_seven, values)
print(list(it))
```

```
>>>
[7, 8, 9, 10]
```

filterfalse

filterfalse, which is the opposite of the filter built-in function, returns all items from an iterator where a predicate function returns False:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = lambda x: x % 2 == 0

filter_result = filter(evens, values)
print('Filter:      ', list(filter_result))

filter_false_result = itertools.filterfalse(evens, values)
print('Filter false:', list(filter_false_result))
```

```
>>>
Filter:       [2, 4, 6, 8, 10]
Filter false: [1, 3, 5, 7, 9]
```

Producing Combinations of Items from Iterators

The itertools built-in module includes a number of functions for producing combinations of items from iterators.

accumulate

accumulate folds an item from the iterator into a running value by applying a function that takes two parameters. It outputs the current accumulated result for each input value:

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum_reduce = itertools.accumulate(values)
print('Sum:   ', list(sum_reduce))

def sum_modulo_20(first, second):
    output = first + second
    return output % 20

modulo_reduce = itertools.accumulate(values, sum_modulo_20)
print('Modulo:', list(modulo_reduce))
```

```
>>>
Sum:    [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
Modulo: [1, 3, 6, 10, 15, 1, 8, 16, 5, 15]
```

This is essentially the same as the reduce function from the functools built-in module, but with outputs yielded one step at a time. By default it sums the inputs if no binary function is specified.

product

product returns the Cartesian product of items from one or more iterators, which is a nice alternative to using deeply nested list comprehensions (see Item 28: "Avoid More Than Two Control Subexpressions in Comprehensions" for why to avoid those):

```
single = itertools.product([1, 2], repeat=2)
print('Single:  ', list(single))

multiple = itertools.product([1, 2], ['a', 'b'])
print('Multiple:', list(multiple))
```

```
>>>
Single:   [(1, 1), (1, 2), (2, 1), (2, 2)]
Multiple: [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

permutations

permutations returns the unique ordered permutations of length *N* with items from an iterator:

```
it = itertools.permutations([1, 2, 3, 4], 2)
print(list(it))
```

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

combinations

combinations returns the unordered combinations of length *N* with unrepeated items from an iterator:

```
it = itertools.combinations([1, 2, 3, 4], 2)
print(list(it))
```

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

combinations_with_replacement

combinations_with_replacement is the same as combinations, but repeated values are allowed:

```
it = itertools.combinations_with_replacement([1, 2, 3, 4], 2)
print(list(it))
```

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

Things to Remember

- ✦ The itertools functions fall into three main categories for working with iterators and generators: linking iterators together, filtering items they output, and producing combinations of items.
- ✦ There are more advanced functions, additional parameters, and useful recipes available in the documentation at help(itertools).

This page intentionally left blank

5 Classes and Interfaces

As an object-oriented programming language, Python supports a full range of features, such as inheritance, polymorphism, and encapsulation. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies.

Python's classes and inheritance make it easy to express a program's intended behaviors with objects. They allow you to improve and expand functionality over time. They provide flexibility in an environment of changing requirements. Knowing how to use them well enables you to write maintainable code.

Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

Python's built-in dictionary type is wonderful for maintaining dynamic internal state over the lifetime of an object. By *dynamic*, I mean situations in which you need to do bookkeeping for an unexpected set of identifiers. For example, say that I want to record the grades of a set of students whose names aren't known in advance. I can define a class to store the names in a dictionary instead of using a predefined attribute for each student:

```
class SimpleGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)
```