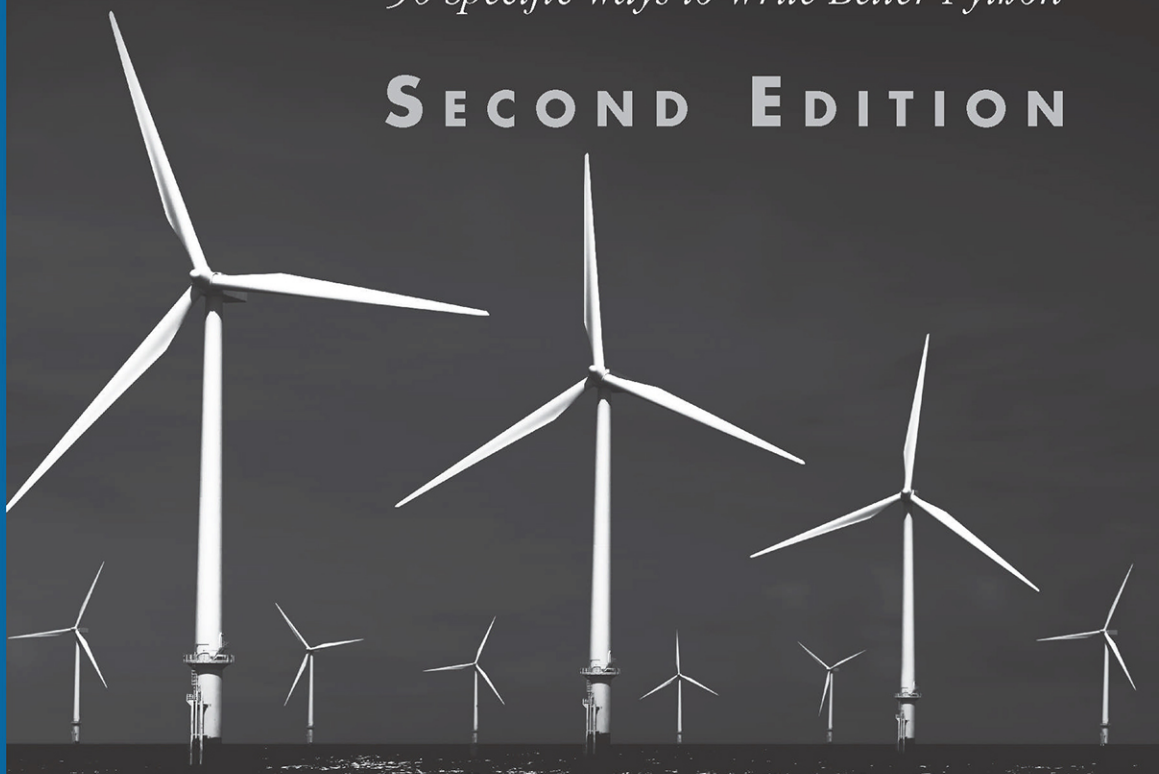
Effective SOFTWARE DEVELOPMENT SERIES

Scott Meyers, Consulting Editor

Effective PYTHON

90 Specific Ways to Write Better Python

SECOND EDITION



Brett Slatkin

Praise for *Effective Python*

"I have been recommending this book enthusiastically since the first edition appeared in 2015. This new edition, updated and expanded for Python 3, is a treasure trove of practical Python programming wisdom that can benefit programmers of all experience levels."

—Wes McKinney, Creator of Python Pandas project, Director of Ursa Labs

"If you're coming from another language, this is your definitive guide to taking full advantage of the unique features Python has to offer. I've been working with Python for nearly twenty years and I still learned a bunch of useful tricks, especially around newer features introduced by Python 3. *Effective Python* is crammed with actionable advice, and really helps define what our community means when they talk about Pythonic code."

—Simon Willison, Co-creator of Django

"I've been programming in Python for years and thought I knew it pretty well. Thanks to this treasure trove of tips and techniques, I've discovered many ways to improve my Python code to make it faster (e.g., using bisect to search sorted lists), easier to read (e.g., enforcing keyword-only arguments), less prone to error (e.g., unpacking with starred expressions), and more Pythonic (e.g., using zip to iterate over lists in parallel). Plus, the second edition is a great way to quickly get up to speed on Python 3 features, such as the walrus operator, f-strings, and the typing module."

—Pamela Fox, Creator of Khan Academy programming courses

"Now that Python 3 has finally become the standard version of Python, it's already gone through eight minor releases and a lot of new features have been added throughout. Brett Slatkin returns with a second edition of *Effective Python* with a huge new list of Python idioms and straightforward recommendations, catching up with everything that's introduced in version 3 all the way through 3.8 that we'll all want to use as we finally leave Python 2 behind. Early sections lay out an enormous list of tips regarding new Python 3 syntaxes and concepts like string and byte objects, f-strings, assignment expressions (and their special nickname you might not know), and catch-all unpacking of tuples. Later sections take on bigger subjects, all of which are packed with things I either didn't know or which I'm always trying to teach to others, including 'Metaclasses and Attributes' (good advice includes 'Prefer Class Decorators over Metaclasses' and also introduces a new magic method '__init_subclass__()' I wasn't familiar with), 'Concurrency' (favorite advice: 'Use Threads for Blocking I/O, but not Parallelism,' but it also covers asyncio and coroutines correctly) and 'Robustness and Performance' (advice given: 'Profile before Optimizing'). It's a joy to go through each section as everything I read is terrific best practice information smartly stated, and I'm considering quoting from this book in the future as it has such great advice all throughout. This is the definite winner for the 'if you only read one Python book this year...' contest.

—Mike Bayer, Creator of SQLAlchemy

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)
```