12
Functional and Reactive Programming Features

In this chapter, we'll look at the following recipes:

- Writing generator functions with the yield statement
- Using stacked generator expressions
- Applying transformations to a collection
- Picking a subset – three ways to filter
- Summarizing a collection – how to reduce
- Combining map and reduce transformations
- Implementing "there exists" processing
- Creating a partial function
- Simplifying complex algorithms with immutable data structures
- Writing recursive generator functions with the yield from statement

Introduction

The idea of **functional programming** is to focus on writing small, expressive functions that perform the required data transformations. Combining functions can often create code which is more succinct and expressive than long strings of procedural statements or the methods of complex, stateful objects. Python allows all three kinds of programming.

Conventional mathematics defines many things as functions. Multiple functions are combined to build up a complex result from previous transformations. For example, we might have two functions, *f(x)* and *g(y)*, that need to be combined to create a useful result:

$$y = f(x)$$

$$z = g(y)$$

Ideally, we can create a composite function from these two functions:

$$z = (g \circ f)(x)$$

Using a composite function, $(g \circ f)$, can help to clarify how a program works. It allows us to take a number of small details and combine them into a larger knowledge chunk.

Since programming often works with collections of data, we'll often be applying a function to a whole collection. This fits nicely with the mathematical idea of a **set builder** or **set comprehension**.

There are three common patterns for applying one function to a set of data:

- **Mapping**: This applies a function to all elements of a collection $\{M(x): x \in C\}$. We apply some function, M, to each item, x, of a larger collection, C.
- **Filtering**: This uses a function to select elements from a collection. $\{x : c \in C$ **if** $F(x)\}$. We use a function, F, to determine whether to pass or reject an item, x, from a larger collection, C.
- **Reducing**: This summarizes a collection. The details vary, but one of the most common reductions is creating a sum of all items, x, in a collection, C: $\sum_{x \in C} x$.

We'll often combine these patterns to create more complex applications. What's important here is that small functions, such as *M(x)* and *F(x)*, are combined via higher-order functions such as mapping and filtering. The combined operation can be sophisticated even though the individual pieces are quite simple.

The idea of **reactive programming** is to have processing rules that are evaluated when the inputs become available or change. This fits with the idea of lazy programming. When we define lazy properties of a class definition, we've created reactive programs.

Reactive programming fits with functional programming because there may be multiple transformations required to react to a change in the input values. Often, this is most clearly expressed as functions that are combined or stacked into a composite function that responds to change. See the *Using properties for lazy attributes* recipe in Chapter 6, *Basics of Classes and Objects*, for some examples of reactive class design.

Writing generator functions with the yield statement

Most of the recipes we've looked at have been designed to work with all of the items in a single collection. The approach has been to use a `for` statement to step through each item within the collection, either mapping the value to a new item, or reducing the collection to some kind of summary value.

Producing a single result from a collection is one of two ways to work with a collection. The alternative is to produce incremental results instead of a single result.

This approach is very helpful in the cases where we can't fit an entire collection in memory. For example, analyzing gigantic web log files is best done in small doses rather than by creating an in-memory collection.

Is there some way to disentangle the collection structure from the processing function? Can we yield results from processing as soon as each individual item is available?

Getting ready

We'll look at some web log data that has date-time string values. We need to parse these to create proper `datetime` objects. To keep things focused in this recipe, we'll use a simplified log produced by Flask.

The entries start out as lines of text that look like this:

```
[2016-05-08 11:08:18,651] INFO in ch09_r09: Sample Message One
[2016-05-08 11:08:18,651] DEBUG in ch09_r09: Debugging
[2016-05-08 11:08:18,652] WARNING in ch09_r09: Something might have gone
wrong
```

We've seen other examples of working with this kind of log in the *Using more complex structures – maps of lists* recipe in Chapter 7, *More Advanced Class Design*. Using REs from the *String parsing with regular expressions* recipe in Chapter 1, *Numbers, Strings, and Tuples*, we can decompose each line to look like the following collection of rows:

```
>>> data = [
...     ('2016-04-24 11:05:01,462', 'INFO', 'module1', 'Sample Message
One'),
...     ('2016-04-24 11:06:02,624', 'DEBUG', 'module2', 'Debugging'),
...     ('2016-04-24 11:07:03,246', 'WARNING', 'module1', 'Something might
have gone wrong')
... ]
```

We can't use ordinary string parsing to convert the complex date-time stamp into something more useful. We can, however, write a generator function which can process each row of the log, producing a more useful intermediate data structure.

A generator function is a function that uses a `yield` statement. When a function has a yield, it builds the results incrementally, yielding each individual value in a way that can be consumed by a client. The consumer might be a `for` statement or it might be another function that needs a sequence of values.

How to do it...

1. This requires the `datetime` module:

   ```
   import datetime
   ```

2. Define a function that processes a source collection:

   ```
   def parse_date_iter(source):
   ```

   We've included the suffix `_iter` as a reminder that this function will be an iterable object, not a simple collection.

3. Include a `for` statement that visits each item in the source collection:

   ```
   for item in source:
   ```

4. The body of the `for` statement can map the item to a new item:

```
date = datetime.datetime.strptime(
    item[0],
    "%Y-%m-%d %H:%M:%S,%f")
new_item = (date,)+item[1:]
```

   In this case, we mapped a single field from string to `datetime` object. The variable `date` is built from the string in `item[0]`.

   Then we mapped the log message three-tuple to a new tuple, replacing the date string with the proper `datetime` object. Since the value of the item is a tuple, we created a singleton tuple with `(date,)` and then concatenated this with the `item[1:]` tuple.

5. Yield the new item with a `yield` statement:

```
yield new_item
```

The whole construct looks like this, properly indented:

```
import datetime
def parse_date_iter(source):
    for item in source:
        date = datetime.datetime.strptime(
            item[0],
            "%Y-%m-%d %H:%M:%S,%f")
        new_item = (date,)+item[1:]
        yield new_item
```

The `parse_date_iter()` function expects an iterable input object. A collection is an example of an iterable object. More importantly, though, other generators are also iterable. We can leverage this to build stacks of generators which process data from other generators.

This function doesn't create a collection. It yields each item, so that the items can be processed individually. The source collection is consumed in small pieces, allowing huge amounts of data to be processed. In some recipes, the data will start out from an in-memory collection. In later recipes, we'll work with data from external files—processing external files benefits the most from this technique.

Here's how we can use this function:

```
>>> from pprint import pprint
>>> from ch08_r01 import parse_date_iter
>>> for item in parse_date_iter(data):
...     pprint(item)
(datetime.datetime(2016, 4, 24, 11, 5, 1, 462000),
 'INFO',
 'module1',
 'Sample Message One')
(datetime.datetime(2016, 4, 24, 11, 6, 2, 624000),
 'DEBUG',
 'module2',
 'Debugging')
(datetime.datetime(2016, 4, 24, 11, 7, 3, 246000),
 'WARNING',
 'module1',
 'Something might have gone wrong')
```

We've used a `for` statement to iterate through the results of the `parse_date_iter()` function, one item at a time. We've used the `pprint()` function to display each item.

We could also collect the items into a proper list using something like this:

```
>>> details = list(parse_date_iter(data))
```

In this example, the `list()` function consumes all of the items produced by the `parse_date_iter()` function. It's essential to use a function such as `list()` or a `for` statement to consume all of the items from the generator. A generator is a relatively passive construct - until data is demanded, it doesn't do any work.

If we don't actively consume the data, we'll see something like this:

```
>>> parse_date_iter(data)
<generator object parse_date_iter at 0x10167ddb0>
```

The value of the `parse_date_iter()` function is a generator. It's not a collection of items, but a function that will produce items on demand.

How it works...

Writing generator functions can change the way we perceive an algorithm. There are two common patterns: mappings and reductions. A mapping transforms each item to a new item, perhaps computing some derived value. A reduction accumulates a summary such as a sum, mean, variance, or hash from the source collection. These can be decomposed into the item-by-item transformation or filter, separate from the overall loop that handles the collection.

Python has a sophisticated construct called an **iterator** which lies at the heart of generators and collections. An iterator will provide each value from a collection while doing all of the internal bookkeeping required to maintain the state of the process. A generator function behaves like an iterator - it provides a sequence of values and maintains its own internal state.

Consider the following common piece of Python code:

```
for i in some_collection:
    process(i)
```

Behind the scenes, something like the following is going on:

```
the_iterator = iter(some_collection)
try:
    while True:
        i = next(the_iterator)
        process(i)
except StopIteration:
    pass
```

Python evaluates the `iter()` function on a collection to create an iterator object for that collection. The iterator is bound to the collection and maintains some internal state information. The code uses `next()` on the iterator to get each value. When there are no more values, the iterator raises the `StopIteration` exception.

Each of Python's collections can produce an iterator. The iterator produced by a `Sequence` or `Set` will visit each item in the collection. The iterator produced by a `Mapping` will visit each key for the mapping. We can use the `values()` method of a mapping to iterate over the values instead of the keys. We can use the `items()` method of a mapping to visit a sequence of `(key, value)` two-tuples. The iterator for a `file` will visit each line in the file.

The iterator concept can also be applied to functions. A function with a `yield` statement is called a **generator function**. It fits the template for an iterator. To do this, the generator returns itself in response to the `iter()` function. In response to the `next()` function, it yields the next value.

When we apply `list()` to a collection or a generator function, the same essential mechanism used by the `for` statement gets the individual values. The `iter()` and `next()` functions are used by `list()` to get the items. The items are then turned into a sequence.

Evaluating `next()` on a generator function is interesting. The generator function is evaluated until it reaches a `yield` statement. This value is the result of `next()`. Each time `next()` is evaluated, the function resumes processing after the `yield` statement and continues to the next `yield` statement.

Here's a small function which yields two objects:

```
>>> def gen_func():
...     print("pre-yield")
...     yield 1
...     print("post-yield")
...     yield 2
```

Here's what happens when we evaluate `next()`. On the generator this function produces:

```
>>> y = gen_func()
>>> next(y)
pre-yield
1
>>> next(y)
post-yield
2
```

The first time we evaluated `next()`, the first `print()` function was evaluated, then the `yield` statement produced a value. The function's processing was suspended and the `>>>` prompt was given. The second time we evaluated the `next()` function, the statements between the two `yield` statements were evaluated. The function was again suspended and a `>>>` prompt will be displayed.

What happens next? We're out of `yield` statements:

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

The `StopIteration` exception is raised at the end of a generator function.

There's more...

The core value of generator functions comes from being able to break complex processing into two parts:

- The transformation or filter to apply
- The source set of data with which to work

Here's an example of using a generator to filter data. In this case, we'll filter the input values and keep only the prime numbers, rejecting all composite numbers.

We can write the processing out as a Python function like this:

```
def primeset(source):
    for i in source:
        if prime(i):
            yield prime
```

For each value in the source, we'll evaluate the `prime()` function. If the result is `true`, we'll yield the source value. If the result is `false`, the source value will be rejected. We can use `primeset()` like this:

```
p_10 = set(primeset(range(2,2000000)))
```

The `primeset()` function will yield individual prime values from a source collection. The source collection will be the integers in the range of 2 to 2 million. The result is a `set` object built from the values provided.

All that's missing from this is the `prime()` function to determine whether a number is prime. We'll leave that as an exercise for the reader.

Mathematically, it's common to see *set builder* or *set comprehension* notation to specify a rule for building one set from another.

We might see something like this:

$P_{10} = \{i: i \in \mathbb{N} \wedge 2 \leq 1 < 2{,}000{,}000 \textbf{ if } P(i)\}$

This tells us that P_{10} is the set of all numbers, i, in the set of natural numbers, $\mathbb{N}$, and between 2 and 2 million if $P(i)$ is `true`. This defines a rule for building a set.

We can write this in Python too:

```
p_10 = {i for i in range(2,2000000) if prime(i)}
```

This is Python notation for the subset of prime numbers. The clauses are rearranged slightly from the mathematical abstraction, but all of the same essential parts of the expression are present.

When we start looking at generator expressions like this, we can see that a great deal of programming fits some common overall patterns:

- **Map**: $\{m(x): x \in S\}$ becomes `(m(x) for x in S)`.
- **Filter**: $\{x: x \in S \textbf{ if } f(x)\}$ becomes `(x for x in S if f(x))`.
- **Reduce**: This is a bit more complex, but common reductions include sums and counts. $\sum_{x \in S} x$ is `sum(x for x in S)`. Other common reductions include finding the maximum or the minimum of a set of data.

We can also write these various higher-level functions using the `yield` statement. Here's the definition of a generic mapping:

```
def map(m, S):
    for s in S:
        yield m(s)
```

This function applies some other function, `m()`, to each data element in the source collection, `S`. The result of the mapping function is yielded as a sequence of result values.

We can write a similar definition for a generic `filter` function:

```
def filter(f, S):
    for s in S:
        if f(s):
            yield s
```

As with the generic mapping, we apply a function, `f()`, to each element in the source collection, `S`. Where the function is `true`, the values are yielded. Where the function is `false`, the values are rejected.

We can use this to create a set of primes like this:

```
p_10 = set(filter(prime, range(2,2000000)))
```

This will apply the `prime()` function to the source range of data. Note that we write just `prime`—without `()` characters—because we're naming the function, not evaluating it. Each individual value will be checked by the `prime()` function. Those that pass will be yielded to be assembled into the final set. Those values which are composite will be rejected and won't wind up in the final set.

See also

- In the *Using stacked generator expressions* recipe, we'll combine generator functions to build complex processing stacks from simple components.
- In the *Applying transformations to a collection* recipe, we'll see how the built-in `map()` function can be used to create complex processing from a simple function and an iterable source of data.
- In the *Picking a subset – three ways to filter* recipe, we'll see how the built-in `filter()` function can also be used to build complex processing from a simple function and an iterable source of data.
- See `https://projecteuler.net/problem=10` for a challenging problem related to prime numbers less than 2 million. Parts of the problem seem obvious. It can be difficult, however, to test all of those numbers for being prime.

Using stacked generator expressions

In the *Writing generator functions with the yield statement* recipe, we created a simple generator function that performed a single transformation on a piece of data. As a practical matter, we often have several functions that we'd like to apply to incoming data.

How can we *stack* or combine multiple generator functions to create a composite function?

Getting ready

We have a spreadsheet that is used to record fuel consumption on a large sailboat. It has rows which look like this:

date	**engine on**	**fuel height**
	engine off	**fuel height**
	Other notes	
10/25/2013	08:24	29
	13:15	27
	calm seas - anchor solomon's island	
10/26/2013	09:12	27
	18:25	22
	choppy - anchor in jackson's creek	

For more background on this data, see the *Slicing and dicing a list* recipe in Chapter 4, *Built-in Data Structures – list, set, dict*.

As a sidebar, we can take the data like this. We'll look at this in detail in the *Reading delimited files with the csv module* recipe in Chapter 8, *Input/Output, Physical Format, and Logical Layout*:

```
>>> from pathlib import Path
>>> import csv
>>> with Path('code/fuel.csv').open() as source_file:
...     reader = csv.reader(source_file)
...     log_rows = list(reader)
>>> log_rows[0]
['date', 'engine on', 'fuel height']
>>> log_rows[-1]
['', "choppy -- anchor in jackson's creek", '']
```

We've used the `csv` module to read the log details. A `csv.reader()` is an iterable object. In order to collect the items into a single list, we applied the `list()` function to the generator function. We printed at the first and last item in the list to confirm that we really have a list-of-lists structure.

We'd like to apply two transformations to this list-of-lists:

- Convert the date and two times into two date-time values
- Merge three rows into one row so that we have a simple organization to the data

If we create a useful pair of generator functions, we can have software that looks like this:

```
total_time = datetime.timedelta(0)
```

```
total_fuel = 0
for row in date_conversion(row_merge(source_data)):
    total_time += row['end_time']-row['start_time']
    total_fuel += row['end_fuel']-row['start_fuel']
```

The combined generator functions, `date_conversion(row_merge(...))`, will yield a sequence of single rows with starting information, ending information, and notes. This structure can easily be summarized or analyzed to create simple statistical correlations and trends.

How to do it...

1. Define an initial reduce operation that combines rows. We have several ways to tackle this. One is to always group three rows together.

   An alternative is to note that column zero has data at the start of a group; it's empty for the next two lines of a group. This gives us a slightly more general approach to creating groups of rows. This is a kind of **head-tail merge** algorithm. We'll collect data and yield the data each time we get to the head of the next group:

   ```
   def row_merge(source_iter):
       group = []
       for row in source_iter:
           if len(row[0]) != 0:
               if group:
                   yield group
               group = row.copy()
           else:
               group.extend(row)
       if group:
           yield group
   ```

   This algorithm uses `len(row[0])` to determine whether this is the head of a group or a row in the tail of the group. In the case of a head row, any previous group is yielded. After that has been consumed, the value of the `group` collection is reset to be the column data from the head row.

The rows in the tail of the group are simply appended to the `group` collection. When the data is exhausted, there will—generally—be one final group in the `group` variable. If there's no data at all, then the final value of `group` will also be a zero-length list, which should be ignored.

We'll address the `copy()` method later. It's essential because we're working with a list of lists data structure and lists are mutable objects. We can write processing which changes the data structures, making some processing awkward to explain.

2. Define the various mapping operations that will be performed on the merged data. These apply to the data in the original row. We'll use separate functions to convert each of the two time columns and merge the times with the date column:

```
        import datetime
        def start_datetime(row):
            travel_date = datetime.datetime.strptime(row[0],
"%m/%d/%y").date()
            start_time = datetime.datetime.strptime(row[1], "%I:%M:%S
%p").time()
            start_datetime = datetime.datetime.combine(travel_date,
start_time)
            new_row = row+[start_datetime]
            return new_row

        def end_datetime(row):
            travel_date = datetime.datetime.strptime(row[0],
"%m/%d/%y").date()
            end_time = datetime.datetime.strptime(row[4], "%I:%M:%S
%p").time()
            end_datetime = datetime.datetime.combine(travel_date, end_time)
            new_row = row+[end_datetime]
            return new_row
```

We'll combine the date in column zero with the time in column one to create a starting `datetime` object. Similarly, we'll combine the date in column zero with the time in column four to create an ending `datetime` object.

These two functions have a lot of overlaps and could be refactored into a single function with the column number as an argument value. For now, however, our goal is to write something that simply works. Refactoring for efficiency can come later.

3. Define mapping operations that apply to the derived data. Columns eight and nine contain the date-time stamps:

```
for starting and ending.def duration(row):
    travel_hours = round((row[10]-row[9]).total_seconds()/60/60, 1)
    new_row = row+[travel_hours]
    return new_row
```

We've used the values created by `start_datetime` and `end_datetime` as inputs. We've computed the delta time, which provides a result in seconds. We converted seconds to hours, which is a more useful unit of time for this set of data.

4. Fold in any filters required to reject or exclude bad data. In this case, we have a header row that must be excluded:

```
def skip_header_date(rows):
    for row in rows:
        if row[0] == 'date':
            continue
        yield row
```

This function will reject any row that has `date` in the first column. The `continue` statement resumes the `for` statement, skipping all other statements in the body; it skips the `yield` statement. All other rows will be passed through this process. The input is an iterable and this generator will yield rows that have not been transformed in any way.

5. Combine the operations. We can either write a sequence of generator expressions or use the built-in `map()` function. Here's how it might look using generator expressions:

```
def date_conversion(source):
    tail_gen = skip_header_date(source)
    start_gen = (start_datetime(row) for row in tail_gen)
    end_gen = (end_datetime(row) for row in start_gen)
    duration_gen = (duration(row) for row in end_gen)
    return duration_gen
```

This operation consists of a series of transformations. Each one does a small transformation on one value from the original collection of data. It's relatively simple to add operations or change operations, since each one is defined independently:

- The `tail_gen` generator yields rows after skipping the first row of the

source

- The `start_gen` generator appends a `datetime` object to the end of each row with the start time built from strings into source columns
- The `end_gen` generator appends a `datetime` object to each row that has the end time built from strings
- The `duration_gen` generator appends a `float` object with the duration of the leg

The output from this overall `date_conversion()` function is a generator. It can be consumed with a `for` statement or a `list` can be built from the items.

How it works...

When we write a generator function, the argument value can be a collection, or it can be another kind of iterable. Since generator functions are iterables, it becomes possible to create a kind of *pipeline* of generator functions.

Each function can embody a small transformation that changes one feature of the input to create the output. We've then wrapped each of these small transformations in generator expressions. Because each transformation is reasonably well isolated from the others, we can make changes to one without breaking the entire processing pipeline.

The processing works incrementally. Each function is evaluated until it yields a single value. Consider this statement:

```
for row in date_conversion(row_merge(data)):
    print(row[11])
```

We've defined a composition of several generators. This composition uses a variety of techniques:

- The `row_merge()` function is a generator which will yield rows of data. In order to yield one row, it will read four lines from the source, assemble a merged row, and yield it. Each time another row is required, it will read three more rows of input to assemble the output row.
- The `date_conversion()` function is a complex generator built from multiple generators.

- `skip_header_date()` is designed to yield a single value. Sometimes it will have to read two values from the source iterator. If an input row has `date` in column zero, the row is skipped. In that case, it will read the second value, getting another row from `row_merge()`; which must, in turn, read three more lines of input to produce a merged line of output. We've assigned the generator to the `tail_gen` variable.
- The `start_gen`, `end_gen`, and `duration_gen` generator expressions will apply relatively simple functions such as `start_datetime()` and `end_datetime()` to each row of its input, yielding rows with more useful data.

The final `for` statement shown in the example will be gathering values from the `date_conversion()` iterator by evaluating the `next()` function repeatedly. Here's the step by step view of what will happen to create the needed result. Note that this works on a very small bit of data—each step makes one small change:

1. The `date_conversion()` function result was the `duration_gen` object. For this to return a value, it needs a row from its source, `end_gen`. Once it has the data, it can apply the `duration()` function and yield the row.
2. The `end_gen` expression needs a row from its source, `start_gen`. It can then apply the `end_datetime()` function and yield the row.
3. The `start_gen` expression needs a row from its source, `tail_gen`. It can then apply the `start_datetime()` function and yield the row.
4. The `tail_gen` expression is simply the generator `skip_header_date()`. This function will read as many rows as required from its source until it finds a row where column zero is not the column header `date`. It yields one non-date row. The source for this is the output from the `row_merge()` function.
5. The `row_merge()` function will read multiple rows from its source until it can assemble a collection of rows that fits the required pattern. It will yield a combined row that has some text in column zero, followed by rows that have no text in column zero. The source for this is a list-of-lists collection of the raw data.
6. The collection of rows will be processed by a `for` statement inside the `row_merge()` function. This processing will implicitly create an iterator for the collection so that each individual row is yielded as needed by the body of the `row_merge()` function.

Each individual row of data will pass through this pipeline of steps. Some stages of the pipeline will consume multiple source rows for a single result row, restructuring the data as it is processed. Other stages consume a single value.

This example relies on concatenating items into a long sequence of values. Items are identified by position. A small change to the order of the stages in the pipeline will alter the positions of the items. There are a number of ways to improve on this that we'll look at next.

What's central to this is that only individual rows are being processed. If the source is a gigantic collection of data, the processing can proceed very quickly. This technique allows a small Python program to process vast volumes of data quickly and simply.

There's more...

In effect, a set of interrelated generators is a kind of composite function. We might have several functions, defined separately like this:

$$y = f(x)$$

$$z = g(y)$$

We can combine them by applying the results of the first function to the second function:

$$z = g(f(x))$$

This can become awkward as the number of functions grows. When we use this pair of functions in multiple places, we break the **Don't Repeat Yourself** (**DRY**) principle. Having multiple copies of this complex expression isn't ideal.

What we'd like to have is a way to create a composite function—something like this:

$$z = (g \circ f)(x)$$

Here, we've defined a new function, $(g \circ f)$, that combines the two original functions into a new, single, composite function. We can now modify this composite to add or change features.

This concept drives the definition of the composite `date_conversion()` function. This function is composed of a number of functions, each of which can be applied to items of collections. If we need to make changes, we can easily write more simple functions and drop them into the pipeline defined by the `date_conversion()` function.

We can see some slight differences among the functions in the pipeline. We have some type conversions. However, the duration calculation isn't really a type conversion. It's a separate computation that's based on the results of the date conversions. If we want to compute fuel use per hour, we'd need to add several more calculations. None of these additional summaries is properly part of date conversion.

We should really break the high-level `data_conversion()` into two parts. We should write another function that does duration and fuel use calculations, named `fuel_use()`. This other function can then wrap `date_conversion()`.

We might aim for something like this:

```
for row in fuel_use(date_conversion(row_merge(data))):
    print(row[11])
```

We now have a very sophisticated computation that's defined in a number of very small and (almost) completely independent chunks. We can modify one piece without having to think deeply about how the other pieces work.

Namespace instead of list

An important change is to stop avoiding the use of a simple list for the data values. Doing computations on `row[10]` is a potential disaster in the making. We should properly convert the input data into some kind of namespace.

A `namedtuple` can be used. We'll look at that in the *Simplifying complex algorithms with immutable data structures* recipe.

A `SimpleNamespace` can, in some ways, further simplify this processing. A `SimpleNamespace` is a mutable object, and can be updated. It's not always the best idea to mutate an object. It has the advantage of being simple, but it can also be slightly more difficult to write tests for state changes in mutable objects.

A function such as `make_namespace()` can provide a set of names instead of positions. This is a generator that must be used after the rows are merged, but before any of the other processing:

```
from types import SimpleNamespace

def make_namespace(merge_iter):
    for row in merge_iter:
        ns = SimpleNamespace(
            date = row[0],
            start_time = row[1],
```

```
                start_fuel_height = row[2],
                end_time = row[4],
                end_fuel_height = row[5],
                other_notes = row[7]
            )
            yield ns
```

This will produce an object that allows us to write `row.date` instead of `row[0]`. This, of course, will change the definitions for the other functions, including `start_datetime()`, `end_datetime()`, and `duration()`.

Each of these functions can emit a new `SimpleNamespace` object instead of updating the list of values that represents each row. We can then write functions that look like this:

```
def duration(row_ns):
    travel_time = row_ns.end_timestamp - row_ns.start_timestamp
    travel_hours = round(travel_time.total_seconds()/60/60, 1)
    return SimpleNamespace(
        **vars(row_ns),
        travel_hours=travel_hours
    )
```

Instead of processing a row as a `list` object, this function processes a row as a `SimpleNamespace` object. The columns have clear and meaningful names such as `row_ns.end_timestamp` instead of the cryptic `row[10]`.

There's a three-part process to building a new `SimpleNamespace` from an old namespace:

1. Use the `vars()` function to extract the dictionary inside the `SimpleNamespace` instance.
2. Use the `**vars(row_ns)` object to build a new namespace based on the old namespace.
3. Any additional keyword parameters such as `travel_hours = travel_hours` provides additional values that will load the new object.

The alternative is to update the namespace and return the updated object:

```
def duration(row_ns):
    travel_time = row_ns.end_timestamp - row_ns.start_timestamp
    row_ns.travel_hours = round(travel_time.total_seconds()/60/60, 1)
    return row_ns
```

This has the advantage of being slightly simpler. The disadvantage is the small consideration that stateful objects can sometimes be confusing. When modifying an algorithm, it's possible to fail to set attributes in the proper order so that lazy (or reactive) programming operates properly.

While stateful objects are common, they should always be viewed as one of two alternatives. An immutable `namedtuple` might be a better choice than a mutable `SimpleNamespace`.

See also

- See the *Writing generator functions with the yield statement* recipe for an introduction to generator functions
- See the *Slicing and dicing a list* recipe in Chapter 4, *Built-in Data Structures – list, set, dict*, for more information on the fuel consumption dataset
- See the *Combining map and reduce transformations* recipe for another way to combine operations

Applying transformations to a collection

In the *Writing generator functions with the yield statement* recipe, we looked at writing a generator function. The examples we saw combined two elements: a transformation and a source of data. They generally look like this:

```
for item in source:
    new_item = some transformation of item
    yield new_item
```

This template for writing a generator function isn't a requirement. It's merely a common pattern. There's a transformation process buried inside a `for` statement. The `for` statement is largely boilerplate code. We can refactor this to make the transformation function explicit and separate from the `for` statement.

In the *Using stacked generator expressions* recipe, we defined a `start_datetime()` function which computed a new `datetime` object from the string values in two separate columns of the source collection of data.

We could use this function in a generator function's body like this:

```
def start_gen(tail_gen):
    for row in tail_gen:
        new_row = start_datetime(row)
        yield new_row
```

This function applies the `start_datetime()` function to each item in a source of data, `tail_gen`. Each resulting row is yielded so that another function or a `for` statement can consume it.

In the *Using stacked generator expressions* recipe, we looked at another way to apply these transformation functions to a larger collection of data. In this example, we used a generator expression. The code looks like this:

```
start_gen = (start_datetime(row) for row in tail_gen)
```

This applies the `start_datetime()` function to each item in a source of data, `tail_gen`. Another function or `for` statement can consume the values available in the `start_gen` iterable.

Both the complete generator function and the shorter generator expression are essentially the same thing with slightly different syntax. Both of these are parallel to the mathematical notion of a *set builder* or *set comprehension*. We could describe this operation mathematically as:

$$s = [S(r): r \in T]$$

In this expression, S is the `start_datetime()` function and T is the sequence of values called `tail_gen`. The resulting sequence is the value of $S(r)$, where each value for r is an element of the set T.

Both generator functions and generator expressions have similar boilerplate code. Can we simplify these?

Getting ready...

We'll look at the web log data from the *Writing generator functions with the yield statement* recipe. This had `date` as a string that we would like to transform into a proper timestamp.

Here's the example data:

```
>>> data = [
...     ('2016-04-24 11:05:01,462', 'INFO', 'module1', 'Sample Message
One'),
...     ('2016-04-24 11:06:02,624', 'DEBUG', 'module2', 'Debugging'),
...     ('2016-04-24 11:07:03,246', 'WARNING', 'module1', 'Something might
have gone wrong')
... ]
```

We can write a function like this to transform the data:

```
import datetime
def parse_date_iter(source):
    for item in source:
        date = datetime.datetime.strptime(
            item[0],
            "%Y-%m-%d %H:%M:%S,%f")
        new_item = (date,)+item[1:]
        yield new_item
```

This function will examine each item in the source using a `for` statement. The value in column zero is a `date` string, which can be transformed into a proper `datetime` object. A new item, `new_item`, is built from the `datetime` object and the remaining items starting with column one.

Because the function uses the `yield` statement to produce results, it's a generator function. We use it with a `for` statement like this:

```
for row in parse_date_iter(data):
    print(row[0], row[3])
```

This statement will gather each value as it's produced by the generator function and print two of the selected values.

The `parse_date_iter()` function has two essential elements combined into a single function. The outline looks like this:

```
for item in source:
    new_item = transformation(item)
    yield new_item
```

The `for` and `yield` statements are largely boilerplate code. The `transformation()` function is a really useful and interesting part of this.

How to do it...

1. Write the transformation function that applies to a single row of the data. This is not a generator, and doesn't use the `yield` statement. It simply revises a single item from a collection:

```
def parse_date(item):
    date = datetime.datetime.strptime(
        item[0],
        "%Y-%m-%d %H:%M:%S,%f")
    new_item = (date,)+item[1:]
    return new_item
```

   This can be used in three ways: statements, expressions, and the `map()` function. Here's the explicit `for...yield` pattern of statements:

```
for item in collection:
    new_item = parse_date(item)
    yield new_item
```

   This uses a `for` statement to process each item in the collection using the isolated `parse_date()` function. The second choice is a generator expression that looks like this:

```
(parse_date(item) for item in data)
```

   This is a generator expression that applies the `parse_date()` function to each item. The third choice is the `map()` function.

2. Use the `map()` function to apply the transformation to the source data.

```
map(parse_date, data)
```

   We provide the name of the function, `parse_date`, without any `()` after the name. We aren't applying the function at this time. We're providing the name of the object to the `map()` function to apply the `parse_date()` function to the iterable source of data, `data`.

   We can use this as follows:

```
for row in map(parse_date, data):
    print(row[0], row[3])
```

The `map()` function creates an iterable object that applies the `parse_date()` function to each item in the data iterable. It yields each individual item. It saves us from having to write a generator expression or a generator function.

How it works...

The `map()` function replaces some common boilerplate code. We can imagine that the definition looks something like this:

```
def map(f, iterable):
    for item in iterable:
        yield f(item)
```

Or, we can imagine that it looks like this:

```
def map(f, iterable):
    return (f(item) for item in iterable)
```

Both of these definitions summarize the core feature of the `map()` function. It's handy shorthand that eliminates some boilerplate code for applying a function to an iterable source of data.

There's more...

In this example, we've used the `map()` function to apply a function that takes a single parameter to each individual item of a single iterable. It turns out that the `map()` function can do a bit more than this.

Consider this function:

```
>>> def mul(a, b):
...     return a*b
```

And these two sources of data:

```
>>> list_1 = [2, 3, 5, 7]
>>> list_2 = [11, 13, 17, 23]
```

We can apply the `mul()` function to pairs drawn from each source of data:

```
>>> list(map(mul, list_1, list_2))
[22, 39, 85, 161]
```

This allows us to merge two sequences of values using different kinds of operators. We can, for example, build a mapping that behaves like the built-in `zip()` function.

Here's a mapping:

```
>>> def bundle(*args):
...     return args
>>> list(map(bundle, list_1, list_2))
[(2, 11), (3, 13), (5, 17), (7, 23)]
```

We needed to define a small helper function, `bundle()`, that takes any number of arguments, and creates a tuple out of them.

Here's the `zip` function for comparison:

```
>>> list(zip(list_1, list_2))
[(2, 11), (3, 13), (5, 17), (7, 23)]
```

See also...

- In the *Using stacked generator expressions* recipe, we looked at stacked generators. We built a composite function from a number of individual mapping operations written as generator functions. We also included a single filter in the stack.

Picking a subset - three ways to filter

In the *Using stacked generator expressions* recipe, we wrote a generator function that excluded some rows from a set of data. We defined a function like this:

```
def skip_header_date(rows):
    for row in rows:
        if row[0] == 'date':
            continue
        yield row
```

When the condition is `true`—`row[0]` is `date`—the `continue` statement will skip the rest of the statements in the body of the `for` statement. In this case, there's only a single statement, `yield row`.

There are two conditions:

- `row[0] == 'date'`: The `yield` statement is skipped; the row is rejected from further processing
- `row[0] != 'date'`: The `yield` statement means that the row will be passed on to the function or statement that's consuming the data

At four lines of code, this seems long-winded. The `for...if...yield` pattern is clearly boilerplate, and only the condition is really material in this kind of construct.

Can we express this more succinctly?

Getting ready...

We have a spreadsheet that is used to record fuel consumption on a large sailboat. It has rows which look like this:

date	**engine on**	**fuel height**
	engine off	**fuel height**
	Other notes	
10/25/2013	08:24	29
	13:15	27
	calm seas - anchor solomon's island	
10/26/2013	09:12	27
	18:25	22
	choppy - anchor in jackson's creek	

For more background on this data, see the *Slicing and dicing a list* recipe.

In the *Using stacked generator expressions* recipe, we defined two functions to reorganize this data. The first combined each three-row group into a single row with a total of eight columns of data:

```
def row_merge(source_iter):
    group = []
    for row in source_iter:
        if len(row[0]) != 0:
            if group:
                yield group
            group = row.copy()
        else:
            group.extend(row)
    if group:
        yield group
```

This is a variation on the **head-tail** algorithm. When `len(row[0]) != 0`, this is the header row for a new group—any previously complete group is yielded, and then the working value of the `group` variable is reset to a fresh, new list based on this header row. A `copy()` is made so that we can avoid mutating the list object later on. When `len(row[0]) == 0`, this is the tail of the group; the row is appended to the working value of the `group` variable. At the end of the source of data, there's generally a complete group that needs to be processed. There's an edge case where there's no data at all; in which case, there's no final group to yield, either.

We can use this function to transform the data from many confusing rows to single rows of useful information:

```
>>> from ch08_r02 import row_merge, log_rows
>>> pprint(list(row_merge(log_rows)))

[['date',
  'engine on',
  'fuel height',
  '',
  'engine off',
  'fuel height',
  '',
  'Other notes',
  ''],
 ['10/25/13',
  '08:24:00 AM',
  '29',
  '',
  '01:15:00 PM',
  '27',
```

```
    '',
    "calm seas -- anchor solomon's island",
    ''],
   ['10/26/13',
    '09:12:00 AM',
    '27',
    '',
    '06:25:00 PM',
    '22',
    '',
    "choppy -- anchor in jackson's creek",
    '']]
```

We see that the first row is just the spreadsheet headers. We'd like to skip this row. We'll create a generator expression to handle the filtering and reject this extra row.

How to do it...

1. Write the predicate function that tests an item to see whether it should be passed through the filter for further processing. In some cases, we'll have to start with a reject rule and then write the inverse to make it into a pass rule:

   ```
   def pass_non_date(row):
       return row[0] != 'date'
   ```

   This can be used in three ways: statements, expressions, and the `filter()` function. Here is an example of an explicit `for...if...yield` pattern of statements for passing rows:

   ```
   for item in collection:
       if pass_non_date(item):
           yield item
   ```

   This uses a `for` statement to process each item in the collection using the filter function. Selected items are yielded. Other items are rejected.

   The second way to use this function is in a generator expression like this:

   ```
   (item for item in data if pass_non_date(item))
   ```

   This generator expressions applies the `filter` function, `pass_non_date()`, to each item. The third choice is the `filter()` function.

2. Use the `filter()` function to apply the function to the source data:

```
filter(pass_non_date, data)
```

We've provided the name of the function, `pass_non_date`. We don't use `()` characters after the function name because this expression doesn't evaluate the function. The `filter()` function will apply the given function to the iterable source of data, `data`. In this case, `data` is a collection, but it can be any iterable, including the results of a previous generator expression. Each item for which the `pass_non_date()` function is `true` will be passed by the filter; all other values are rejected.

We can use this as follows:

```
for row in filter(pass_non_date, row_merge(data)):
    print(row[0], row[1], row[4])
```

The `filter()` function creates an iterable object that applies the `pass_non_date()` function as a rule to pass or reject each item in the `row_merge(data)` iterable. It yields the rows that don't have `date` in column zero.

How it works...

The `filter()` function replaces some common boilerplate code. We can imagine that the definition looks something like this:

```
def filter(f, iterable):
    for item in iterable:
        if f(item):
            yield f(item)
```

Or, we can imagine that it looks like this:

```
def filter(f, iterable):
    return (item for item in iterable if f(item))
```

Both of these definitions summarize the core feature of the `filter()` function: some data is passed and some data is rejected. This is a handy shorthand that eliminates some boilerplate code for applying a function to an iterable source of data.

There's more...

Sometimes it's difficult to write a simple rule to pass data. It may be clearer if we write a rule to reject data. For example, this might make more sense:

```
def reject_date(row):
    return row[0] == 'date'
```

We can use a reject rule in a number of ways. Here's a `for...if...continue...yield` pattern of statements. This will use continue to skip the rejected rows, and yield the remaining rows:

```
for item in collection:
    if reject_date(item):
        continue
    yield item
```

We can also use this variation. For some programmers, the not reject concept can become confusing. It might seem like a double negative:

```
for item in collection:
    if not reject_date(item):
        yield item
```

We can also use a generator expression like this:

```
(item for item in data if not reject_date(item))
```

We can't, however, easily use the `filter()` function with a rule that's designed to reject data. The `filter()` function is designed to work with pass rules only.

We have two essential choices for dealing with this kind of logic. We can wrap the logic in another expression, or we use a function from the `itertools` module. When it comes to wrapping, we have two further choices. We can wrap a reject function to create a pass function from it. We can use something like this:

```
def pass_date(row):
    return not reject_date(row)
```

This makes it possible to create a simple reject rule and use it in the `filter()` function. Another way to wrap the logic is to create a `lambda` object:

```
 filter(lambda item: not reject_date(item), data)
```

The `lambda` function is a small, anonymous function. It's a function that's been reduced to just two elements: the parameter list and a single expression. We've wrapped the `reject_date()` function to create a kind of `not_reject_date` function via the `lambda` object.

In the `itertools` module, we use the `filterfalse()` function. We can import `filterfalse()` and use this instead of the built-in `filter()` function.

See also...

- In the *Using stacked generator expressions* recipe, we placed a function like this in a stack of generators. We built a composite function from a number of individual mapping and filtering operations written as generator functions.

Summarizing a collection – how to reduce

In the introduction to this chapter, we noted that there are three common processing patterns: map, filter, and reduce. We've seen examples of mapping in the *Applying transformations to a collection* recipe, and examples of filtering in the *Picking a subset – three ways to filter* recipe. It's relatively easy to see how these become very generic operations.

Mapping applies a simple function to all elements of a collection. $\{M(x): x \in C\}$ applies a function, M, to each item, x, of a larger collection, C. In Python, it can look like this:

```
(M(x) for x in C)
```

Or, we can use the built-in `map()` function to remove the boilerplate and simplify it to this:

```
map(M, c)
```

Similarly, filtering uses a function to select elements from a collection. $\{x: x \in C$ **if** $F(x)\}$ uses a function, F, to determine whether to pass or reject an item, x, from a larger collection, C. We can express this in a variety of ways in Python, one of which is like this:

```
filter(F, c)
```

This applies a predicate function, `F()`, to a collection, `c`.

The third common pattern is reduction. In the *Designing classes with lots of processing* and *Extending a collection: a list that does statistics* recipes, we looked at class definitions that computed a number of statistical values. These definitions relied—almost exclusively—on the built-in `sum()` function. This is one of the more common reductions.

Can we generalize summation in a way that allows us to write a number of different kinds of reductions? How can we define the concept of reduction in a more general way?

Getting ready

One of the most common reductions is the sum. Other reductions include a product, minimum, maximum, average, variance, and even a simple count of values.

Here's a way to think of the mathematical definition of the sum function, +, applied to values in a collection, *C*:

$$\sum_{c_i \in C} c_i = c_0 + c_1 + c_2 + \ldots + c_n$$

We've expanded the definition of sum by inserting the + operator into the sequence of values, $C = c_0, c_1, c_2, \ldots, c_n$. This idea of *folding* in the + operator captures the meaning of the built-in `sum()` function.

Similarly, the definition of product looks like this:

$$\prod_{c_i \in C} c_i = c_0 \times c_1 \times c_2 \times \ldots \times c_n$$

Here, too, we've performed a different *fold* on a sequence of values. Expanding a reduction by folding involves two items: a binary operator and a base value. For sum, the operator was + and the base value is zero. For product, the operator is × and the base value is one.

We could define a generic higher-level function, $F_{(\diamond, \perp)}$, that captures the ideal of a fold. The fold function definition includes a placeholder for an operator, $\diamond$, and a placeholder for a base value, $\perp$. The function's value for a given collection, C, can be defined with this recursive rule:

$$F_{(\diamond,\perp)}(C) = \begin{cases} \perp & \text{if } C = \varnothing \\ F_{(\diamond,\perp)}(C_{0..n-1}) \diamond C_{n-1} & \text{if } C \neq \varnothing \end{cases}$$

If the collection, C, is empty, the value is the base value, $\perp$. When defining `sum()`, the base value would be zero. If C is not empty, then we'll first compute the fold of everything but the last value in the collection, $F_{\diamond, \perp}(C_{0..n-1})$. Then we'll apply the operator—for example, addition—between the previous fold result and the final value in the collection, C_{n-1}. For `sum()`, the operator is +.

We've used the notation $C_{0..n}$ in the Pythonic sense of an open-ended range. The values at indices 0 to n-1 are included, but the value of index n is not included. This means that $C_{0..0} = \varnothing$: there are no elements in this range $C_{0..0}$.

This definition is called a **fold left** operation because the net effect of this definition is to perform the underlying operations from left to right in the collection. This could be changed to also define a **fold right** operation. Since Python's `reduce()` function is a fold left, we'll stick with that.

We'll define a `prod()` function that can be used to compute factorial values:

$$n! = \prod_{1 \leq x < n+1} x$$

The value of n factorial is the product of all of the numbers between 1 and n inclusive. Since Python uses half-open ranges, it's a little more Pythonic to use or define a range using $1 \leq x < n + 1$. This definition fits the built-in `range()` function better.

Using the fold operator that we defined earlier, we have this. We've defined a fold (or reduce) using an operator of multiplication, *, and a base value of one:

$$n! = \prod_{1 \leq x < n+1} x = F_{\times,1}(i : 1 \leq i < n+1)$$

The idea of folding is the generic concept that underlies Python's concept of `reduce()`. We can apply this to many algorithms, potentially simplifying the definition.

How to do it...

1. Import the `reduce()` function from the `functools` module:

```
>>> from functools import reduce
```

2. Pick the operator. For sum, it's `+`. For product, it's `*`. These can be defined in a variety of ways. Here's the long version. Other ways to define the necessary binary operator will be shown later:

```
>>> def mul(a, b):
...     return a * b
```

3. Pick the base value required. For sum, it's zero. For product, it's one. This allows us to define a `prod()` function that computes a generic product:

```
>>> def prod(values):
...     return reduce(mul, values, 1)
```

4. For factorial, we need to define the sequence of values that will be reduced:

```
range(1, n+1)
```

   Here's how this works with the `prod()` function:

```
>>> prod(range(1, 5+1))
120
```

Here's the whole factorial function:

```
>>> def factorial(n):
...     return prod(range(1, n+1))
```

Here's the number of ways that a 52-card deck can be arranged. This is the value *52!*:

```
>>> factorial(52)
80658175170943878571660636856403766975289505440883277824000000000000
```

There are a lot of a ways a deck can be shuffled.

How many 5-card hands are possible? The binomial calculation uses factorial:

$$\boxed{\binom{52}{5} = \frac{52!}{5!(52-5)!}}$$

```
>>> factorial(52)//(factorial(5)*factorial(52-5))
2598960
```

For any given shuffle, there are about 2.6 million different possible poker hands. (And yes, this is a terribly inefficient way to compute the binomial.)

How it works...

The `reduce()` function behaves as though it had this definition:

```
def reduce(function, iterable, base):
    result = base
    for item in iterable:
        result = function(result, item)
    return result
```

This will iterate through the values from left to right. It will apply the given binary function between the previous set of values and the next item from the iterable collection.

When we look at the *Recursive functions and Python's stack limits* recipe, we can see that the recursive definition of fold can be optimized to this `for` statement.

There's more...

When designing a `reduce()` function, we need to provide a binary operator. There are three ways to define the necessary binary operator. We used a complete function definition like this:

```
def mul(a, b):
    return a * b
```

There are two other choices. We can use a `lambda` object instead of a complete function:

```
>>> add = lambda a, b: a + b
>>> mul = lambda a, b: a * b
```

A `lambda` function is an anonymous function boiled down to just two essential elements: the parameters and the return expression. There are no statements inside a lambda, only a single expression. In this case, the expression simply uses the desired operator.

We can use it like this:

```
>>> def prod2(values):
...     return reduce(lambda a, b: a*b, values, 1)
```

This provides the multiplication function as a `lambda` object without the overhead of a separate function definition.

We can also import the definition from the `operator` module:

```
from operator import add, mul
```

This works nicely for all of the built-in arithmetic operators.

Note that logical reductions using the logic operators **AND** and **OR** are a little different from other arithmetic reductions. These operators short-circuit: once the value is `false`, an **and-reduce** can stop processing. Similarly, once the value is `True`, an **or-reduce** can stop processing. The built-in functions `any()` and `all()` embody this nicely. The short-circuit feature is difficult to capture using the built-in `reduce()`.

Maxima and minima

How can we use `reduce()` to compute a maximum or minimum? This is a little more complex because there's no trivial base value that can be used. We cannot start with zero or one because these values might be outside the range of values being minimized or maximized.

Also, the built-in `max()` and `min()` must raise an exception for an empty sequence. These functions can't fit perfectly with the way the `sum()` function and `reduce()` functions work.

We have to use something like this to provide the expected feature set:

```
def mymax(sequence):
    try:
        base = sequence[0]
        max_rule = lambda a, b: a if a > b else b
```

```
            reduce(max_rule, sequence, base)
        except IndexError:
            raise ValueError
```

This function will pick the first value from the sequence as a base value. It creates a `lambda` object, named `max_rule`, which selects the larger of the two argument values. We can then use this base value located in the data, and the `lambda` object. The `reduce()` function will then locate the largest value in a non-empty collection. We've captured the `IndexError` exception so that an empty collection will raise a `ValueError` exception.

This example shows how we can invent a more complex or sophisticated minimum or maximum function that is still based on the built-in `reduce()` function. The advantage of this is replacing the boilerplate `for` statement when reducing a collection to a single value.

Potential for abuse

Note that a fold (or `reduce()` as it's called in Python) can be abused, leading to poor performance. We have to be cautious about simply using a `reduce()` function without thinking carefully about what the resulting algorithm might look like. In particular, the operator being folded into the collection should be a simple process such as adding or multiplying. Using `reduce()` changes the complexity of an $O(1)$ operation into $O(n)$.

Imagine what would happen if the operator being applied during the reduction involved a sort over a collection. A complex operator—with $O(n \log n)$ complexity—being used in a `reduce()` would change the complexity of the overall `reduce()` to $O(n^2 \log n)$.

Combining map and reduce transformations

In the other recipes in this chapter, we've been looking at map, filter, and reduce operations. We've looked at each of these in isolation:

- The *Applying transformations to a collection* recipe shows the `map()` function
- The *Picking a subset – three ways to filter* recipe shows the `filter()` function
- The *Summarizing a collection – how to reduce* recipe shows the `reduce()` function

Many algorithms will involve combinations of functions. We'll often use mapping, filtering, and a reduction to produce a summary of available data. Additionally, we'll need to look at a profound limitation of working with iterators and generator functions. Namely this limitation:

An iterator can only produce values once.

If we create an iterator from a generator function and a collection data, the iterator will only produce the data one time. After that, it will appear to be an empty sequence.

Here's an example:

```
>>> typical_iterator = iter([0, 1, 2, 3, 4])
>>> sum(typical_iterator)
10
>>> sum(typical_iterator)
0
```

We created an iterator over a sequence of values by manually applying the `iter()` function to a literal list object. The first time that the `sum()` function used the value of `typical_iterator`, it consumed all five values. The next time we tried to apply any function to the `typical_iterator`, there will be no more values to be consumed - the iterator appears empty.

This basic one-time-only restriction drives some of the design considerations when working with multiple kinds of generator functions in conjunction with map, filter, and reduce. We'll often need to cache intermediate results so that we can perform multiple reductions on the data.

Getting ready

In the *Using stacked generator expressions* recipe, we looked at data that required a number of processing steps. We merged rows with a generator function. We filtered some rows to remove them from the resulting data. Additionally, we applied a number of mappings to the data to convert dates and times to more useful information.

We'd like to supplement this with two more reductions to get some average and variance information. These statistics will help us understand the data more fully.

We have a spreadsheet that is used to record fuel consumption on a large sailboat. It has rows which look like this:

date	**engine on**	**fuel height**
	engine off	**fuel height**
	Other notes	
10/25/2013	08:24	29
	13:15	27
	calm seas - anchor solomon's island	
10/26/2013	09:12	27
	18:25	22
	choppy - anchor in jackson's creek	

The initial processing was a sequence of operations to change the organization of the data, filter out the headings, and compute some useful values.

How to do it...

1. Start with the goal. In this case, we'd like a function we can use like this:

```
>>> round(sum_fuel(clean_data(row_merge(log_rows))), 3)
7.0
```

   This shows a three-step pattern for this kind of processing. These three steps will define our approach to creating the various parts of this reduction:

   1. First, transform the data organization. This is sometimes called normalizing the data. In this case, we'll use a function called `row_merge()`. See the *Using stacked generator expressions* recipe for more information on this.
   2. Second, use mapping and filtering to clean and enrich the data. This is defined as a single function, `clean_data()`.

3. Finally, reduce the data to a sum with `sum_fuel()`. There are a variety of other reductions that make sense. We might compute averages, or sums of other values. There are a lot of reductions we might want to apply.

2. If needed, define the data structure normalization function. This almost always has to be a generator function. A structural change can't be applied via `map()`:

```
from ch08_r02 import row_merge
```

   As shown in the *Using stacked generator expressions* recipe, this generator function will restructure the data from three rows per each leg of the voyage to one row per leg. When all of the columns are in a single row, the data is much easier to work with.

3. Define the overall data cleansing and enrichment data function. This is a generator function that's built from simpler functions. It's a stack of `map()` and `filter()` operations that will derive data from the source fields:

```
def clean_data(source):
    namespace_iter = map(make_namespace, source)
    fitered_source = filter(remove_date, namespace_iter)
    start_iter = map(start_datetime, fitered_source)
    end_iter = map(end_datetime, start_iter)
    delta_iter = map(duration, end_iter)
    fuel_iter = map(fuel_use, delta_iter)
    per_hour_iter = map(fuel_per_hour, fuel_iter)
    return per_hour_iter
```

   Each of the `map()` and `filter()` operations involves a small function to do a single conversion or computation on the data.

4. Define the individual functions that are used for cleansing and deriving additional data.
5. Convert the merged rows of data into a `SimpleNamespace`. This will allow us to use names such as `start_time` instead of `row[1]`:

```
from types import SimpleNamespace
def make_namespace(row):
    ns = SimpleNamespace(
        date = row[0],
        start_time = row[1],
        start_fuel_height = row[2],
        end_time = row[4],
        end_fuel_height = row[5],
        other_notes = row[7]
```

```
            )
            return ns
```

This function builds a `SimpleNamspace` from selected columns of the source data. Columns three and six were omitted because they were always zero-length strings, `''`.

6. Here's the function used by the `filter()` to remove the heading rows. If needed, this can be expanded to remove blank lines or other bad data from the source. The idea is to remove bad data as soon as possible in the processing:

```
        def remove_date(row_ns):
            return not(row_ns.date == 'date')
```

7. Convert data to a usable form. First, we'll convert strings to dates. The next two functions depend on this `timestamp()` function that converts a `date` string from one column plus a `time` string from another column into a proper `datetime` instance:

```
        import datetime
        def timestamp(date_text, time_text):
            date = datetime.datetime.strptime(date_text, "%m/%d/%y").date()
            time = datetime.datetime.strptime(time_text, "%I:%M:%S
%p").time()
            timestamp = datetime.datetime.combine(date, time)
            return timestamp
```

This allows us to do simple date calculations based on the `datetime` library. In particular, subtracting two timestamps will create a `timedelta` object that has the exact number of seconds between any two dates.

Here's how we'll use this function to create a proper timestamp for the start of the leg and the end of the leg:

```
        def start_datetime(row_ns):
            row_ns.start_timestamp = timestamp(row_ns.date,
row_ns.start_time)
            return row_ns

        def end_datetime(row_ns):
            row_ns.end_timestamp = timestamp(row_ns.date, row_ns.end_time)
            return row_ns
```

Both of these functions will add a new attribute to the `SimpleNamespace` and also return the namespace object. This allows these functions to be used in a stack of `map()` operations. We can also rewrite these functions to replace the mutable `SimpleNamespace` with an immutable `namedtuple()` and still preserve the stack of `map()` operations.

8. Compute derived time data. In this case, we can compute a duration too. Here's a function which must be performed after the previous two:

```
def duration(row_ns):
    travel_time = row_ns.end_timestamp - row_ns.start_timestamp
    row_ns.travel_hours = round(travel_time.total_seconds()/60/60,
1)
    return row_ns
```

This will convert the difference in seconds into a value in hours. It will also round to the nearest tenth of an hour. Any more accuracy than this is largely noise. The departure and arrival times are (generally) off by at least a minute; they depend on when the skipper remembered to look at her watch. In some cases, she may have estimated the time.

9. Compute other metrics that are needed for the analyses. This includes creating the height values that are converted to float numbers. The final calculation is based on two other calculated results:

```
def fuel_use(row_ns):
    end_height = float(row_ns.end_fuel_height)
    start_height = float(row_ns.start_fuel_height)
    row_ns.fuel_change = start_height - end_height
    return row_ns

def fuel_per_hour(row_ns):
    row_ns.fuel_per_hour = row_ns.fuel_change/row_ns.travel_hours
    return row_ns
```

The fuel per hour calculation depends on the entire preceding stack of calculations. The travel hours comes from the start and end timestamps which are computed separately.

How it works...

The idea is to create a composite operation that follows a common template:

1. Normalize the structure: This often requires a generator function to read data in one structure and yield data in a different structure.
2. Filter and cleanse: This may involve a simple filter as shown in this example. We'll look at more complex filters later.
3. Derive data via mappings or via lazy properties of class definitions: A class with lazy properties is a reactive object. Any change to the source property should cause changes to the computed properties.

In some cases, we may want to combine the basic facts with other dimensional descriptions. For example, we might need to look up reference data, or decode coded fields.

Once we've done the preliminary steps, we have data which is usable for a variety of analyses. Many times, this is a reduce operation. The initial example computes a sum of fuel use. Here are two other examples:

```
from statistics import *
def avg_fuel_per_hour(iterable):
    return mean(row.fuel_per_hour for row in iterable)
def stdev_fuel_per_hour(iterable):
    return stdev(row.fuel_per_hour for row in iterable)
```

These functions apply the `mean()` and `stdev()` functions to the `fuel_per_hour` attribute of each row of the enriched data.

We might use this as follows:

```
>>> round(avg_fuel_per_hour(
...     clean_data(row_merge(log_rows))), 3)
0.48
```

We've used the `clean_data(row_merge(log_rows))` mapping pipeline to cleanse and enrich the raw data. Then we applied a reduction to this data to get the value we're interested in.

We now know that our 30" tall tank is good for about 60 hours of motoring. At 6 knots, we can go about 360 nautical miles on a full tank of fuel.

There's more...

As we noted, we can only perform one reduction on an iterable source of data. If we want to compute several averages, or the average and the variance, we'll need to use a slightly different pattern.

In order to compute multiple summaries of the data, we'll need to create a sequence object of some kind that can be summarized repeatedly:

```
data = tuple(clean_data(row_merge(log_rows)))
m = avg_fuel_per_hour(data)
s = 2*stdev_fuel_per_hour(data)
print("Fuel use {m:.2f} ±{s:.2f}".format(m=m, s=s))
```

Here, we've created a `tuple` from the cleaned and enriched data. This `tuple` will produce an iterable, but unlike a generator function, it can produce this iterable many times. We can compute two summaries by using the `tuple` object.

This design involves a large number of transformations of source data. We've built it using a stack of map, filter, and reduce operations. This provides a great deal of flexibility.

The alternative approach is to create a class definition. A class can be designed with lazy properties. This would create a kind of reactive design embodied in a single block of code. See the *Using properties for lazy attributes* recipe for examples of this.

We can also use the `tee()` function in the `itertools` module for this kind of processing:

```
from itertools import tee
data1, data2 = tee(clean_data(row_merge(log_rows)), 2)
m = avg_fuel_per_hour(data1)
s = 2*stdev_fuel_per_hour(data2)
```

We've used `tee()` to create two clones of the iterable output from `clean_data(row_merge(log_rows))`. We can use these two clones to compute a mean and a standard deviation.

See also

- We looked at how to combine mapping and filtering in the *Using stacked generator expressions* recipe.
- We looked at lazy properties in the *Using properties for lazy attributes* recipe. Also, this recipe looks at some important variations on map-reduce processing.

Implementing "there exists" processing

The processing patterns we've been looking at can all be summarized with the quantifier *for all*. It's been an implicit part of all of the processing definitions:

- **Map**: For all items in the source, apply the map function. We can use the quantifier to make this explicit: $\{M(x) \forall x: x \in C\}$
- **Filter**: For all items in the source, pass those for which the filter function is `true`. Here also, we've used the quantifier to make this explicit. We want all values, x, from a set, C, if some function, $F(x)$, is `true`: $\{x \forall x: x \in C \textbf{ if } F(x)\}$
- **Reduce**: For all items in the source, use the given operator and base value to compute a summary. The rule for this is a recursion that clearly works for all values of the source collection or iterable: $$F_{\Diamond,\perp}\left(C_{0..n}\right) = \begin{cases} \perp & \text{if } C = \varnothing \\ F_{\Diamond,\perp}\left(C_{0..n-1}\right) \Diamond C_{n-1} & \text{if } C \neq \varnothing \end{cases}.$$

We've used the notation $C_{0..n}$ in the Pythonic sense of an open-ended range. Values with index positions of 0 and $n-1$ are included, but the value at index position n is not included. This means that there are no elements in this range.

What's more important is that $C_{0..n-1} \cup C_{n-1} = C$. That is, when we take the last item from the range, no items are lost—we're always processing all the items in the collection. Also, we aren't processing item C_{n-1} twice. It's not part of the $C_{0..n-1}$ range, but it is a standalone item C_{n-1}.

How can we write a process using generator functions that stops when the first value matches some predicate? How do we avoid *for all* and quantify our logic with *there exists*?

Getting ready

There's another quantifier that we might need—*there exists*, $\exists$. Let's look at an example of an existence test.

We might want to know whether a number is prime or composite. We don't need all of the factors of a number to know it's not prime. It's sufficient to show that a factor exists to know that a number is not prime.

We can define a prime predicate, *P(n)*, like this:

$$P(n) = \neg\exists i: 2 \leq i < n \textbf{ if } n \bmod i = 0$$

A number, *n*, is prime if there does not exist a value of *i* (between 2 and the number) that divides the number evenly. We can move the negation around and rephrase this as follows:

$$\neg P(n) = \exists i: 2 \leq i < n \textbf{ if } n \bmod i = 0$$

A number, *n*, is composite (non-prime) if there exists a value, *i*, between 2 and the number itself, that divides the number evenly. We don't need to know **all** such values. The existence of one value that satisfies the predicate is sufficient.

Once we've found such a number, we can break early from any iteration. This requires the `break` statement inside `for` and `if` statements. Because we're not processing all values, we can't easily use a higher-order function such as `map()`, `filter()`, or `reduce()`.

How to do it...

1. Define a generator function template that will skip items until the required one is found. This will yield only one value that passes the predicate test:

```
def find_first(predicate, iterable):
    for item in iterable:
        if predicate(item):
            yield item
            break
```

2. Define a predicate function. For our purposes, a simple `lambda` object will do. Also, a lambda allows us to work with a variable bound to the iteration and a variable that's free from the iteration. Here's the expression:

```
lambda i: n % i == 0
```

   We're relying on a non-local value, `n`, in this lambda. This will be *global* to the lambda, but still local to the overall function. If `n % i` is `0`, then `i` is a factor of `n`, and `n` is not prime.

3. Apply the function with the given range and predicate:

```
import math
def prime(n):
    factors = find_first(
        lambda i: n % i == 0,
        range(2, int(math.sqrt(n)+1)) )
    return len(list(factors)) == 0
```

If the `factors` iterable has an item, then `n` is composite. Otherwise, there are no values in the `factors` iterable, which means `n` is a prime number.

As a practical matter, we don't need to test every single number between two and `n` to see whether `n` is prime. It's only necessary to test values, `i`, such that $2 \le i < \sqrt{n}$.

How it works...

In the `find_first()` function, we introduce a `break` statement to stop processing the source iterable. When the `for` statement stops, the generator will reach the end of the function, and return normally.

The process which is consuming values from this generator will be given the `StopIteration` exception. This exception means the generator will produce no more values. The `find_first()` function raises as an exception, but it's not an error. It's the normal way to signal that an iterable has finished processing the input values.

In this case, the signal means one of two things:

- If a value has been yielded, the value is a factor of `n`
- If no value was yielded, then `n` is prime

This small change of breaking early from the `for` statement makes a dramatic difference in the meaning of the generator function. Instead of processing **all** values from the source, the `find_first()` generator will stop processing as soon as the predicate is `true`.

This is different from a filter, where all of the source values will be consumed. When using the `break` statement to leave the `for` statement early, some of the source values may not be processed.

There's more...

In the `itertools` module, there is an alternative to this `find_first()` function. The `takewhile()` function uses a predicate function to keep taking values from the input. When the predicate becomes `false`, then the function stops processing values.

We can easily change the lambda from `lambda i: n % i == 0` to `lambda i: n % i != 0`. This will allow the function to take values while they are not factors. Any value that is a factor will stop the processing by ending the `takewhile()` process.

Let's look at two examples. We'll test `13` for being prime. We need to check numbers in the range. We'll also test `15` for being prime:

```
>>> from itertools import takewhile
>>> n = 13
>>> list(takewhile(lambda i: n % i != 0, range(2, 4)))
[2, 3]
>>> n = 15
>>> list(takewhile(lambda i: n % i != 0, range(2, 4)))
[2]
```

For a prime number, all of the test values pass the `takewhile()` predicate. The result is a list of non-factors of the given number, *n*. If the set of non-factors is the same as the set of values being tested, then `n` is prime. In the case of `13`, both collections of values are `[2, 3]`.

For a composite number, some values pass the `takewhile()` predicate. In this example, `2` is not a factor of `15`. However, `3` is a factor; this does not pass the predicate. The collection of non-factors, `[2]`, is not the same as the set of values collection that was tested, `[2, 3]`.

We wind up with a function that looks like this:

```
def prime_t(n):
    tests = set(range(2, int(math.sqrt(n)+1)))
    non_factors = set(
        takewhile(
            lambda i: n % i != 0,
            tests
        )
    )
    return tests == non_factors
```

This creates two intermediate set objects, `tests` and `non_factors`. If all of the tested values are not factors, the number is prime. The function shown previously, based on `find_first()` only creates one intermediate list object. That list will have at most one member, making the data structure much smaller.

The itertools module

There are a number of additional functions in the `itertools` module that we can use to simplify complex map-reduce applications:

- `filterfalse()`: It is the companion to the built-in `filter()` function. It inverts the predicate logic of the `filter()` function; it rejects items for which the predicate is `true`.
- `zip_longest()`: It is the companion to the built-in `zip()` function. The built-in `zip()` function stops merging items when the shortest iterable is exhausted. The `zip_longest()` function will supply a given fill value to pad short iterables to match the longest.
- `starmap()`: It is a modification to the essential `map()` algorithm. When we perform `map(function, iter1, iter2)`, then an item from each iterable is provided as two positional arguments to the given function. The `starmap()` expects an iterable to provide a tuple that contains the argument values. In effect:

```
map = starmap(function, zip(iter1, iter2))
```

There are still others that we might use, too:

- `accumulate()`: This function is a variation on the built-in `sum()` function. This will yield each partial total that's produced before reaching the final sum.
- `chain()`: This function will combine iterables in order.
- `compress()`: This function uses one iterable as a source of data and the other as a source of selectors. When the item from the selector is true, the corresponding data item is passed. Otherwise, the data item is rejected. This is an item-by-item filter based on true-false values.
- `dropwhile()`: While the predicate to this function is `true`, it will reject values. Once the predicate becomes `false`, it will pass all remaining values. See `takewhile()`.
- `groupby()`: This function uses a key function to control the definition of groups. Items with the same key value are grouped into separate iterators. For the results to be useful, the original data should be sorted into order by the keys.

- `islice()`: This function is like a slice expression, except it applies to an iterable, not a list. Where we use `list[1:]` to discard the first row of a list, we can use `islice(iterable, 1)` to discard the first item from an iterable.
- `takewhile()`: While the predicate is `true`, this function will pass values. Once the predicate becomes `false`, stop processing any remaining values. See `dropwhile()`.
- `tee()`: This splits a single iterable into a number of clones. Each clone can then be consumed separately. This is a way to perform multiple reductions on a single iterable source of data.

Creating a partial function

When we look at functions such as `reduce()`, `sorted()`, `min()`, and `max()`, we see that we'll often have some *permanent* argument values. For example, we might find a need to write something like this in several places:

```
reduce(operator.mul, ..., 1)
```

Of the three parameters to `reduce()`, only one - the iterable to process - actually changes. The operator and the base value arguments are essentially fixed at `operator.mul` and `1`.

Clearly, we can define a whole new function for this:

```
def prod(iterable):
    return reduce(operator.mul, iterable, 1)
```

However, Python has a few ways to simplify this pattern so that we don't have to repeat the boilerplate `def` and `return` statements.

How can we define a function that has some parameters provided in advance?

Note that the goal here is different from providing default values. A partial function doesn't provide a way to override the defaults. Instead, we want to create as many partial functions as we need, each with specific parameters bound in advance.

Getting ready

Some statistical modeling is done with standard scores, sometimes called **z-scores**. The idea is to standardize a raw measurement onto a value that can be easily compared to a normal distribution, and easily compared to related numbers that are measured in different units.

The calculation is this:

$$z = (x - \mu)/\sigma$$

Here, x is the raw value, μ is the population mean, and σ is the population standard deviation. The value z will have a mean of 0 and a standard deviation of 1. This makes it particularly easy to work with.

We can use this value to spot **outliers** - values which are suspiciously far from the mean. We expect that (about) 99.7% of our z values will be between -3 and +3.

We could define a function like this:

```
def standarize(mean, stdev, x):
    return (x-mean)/stdev
```

This `standardize()` function will compute a z-score from a raw score, x. This function has two kinds of parameters:

- The values for `mean` and `stdev` are essentially fixed. Once we've computed the population values, we'll have to provide them to the `standardize()` function over and over again.
- The value for `x` is more variable.

Let's say we've got a collection of data samples in big blocks of text:

```
text_1 = '''10  8.04
8       6.95
13      7.58
...
5       5.68
'''
```

We've defined two small functions to convert this data to pairs of numbers. The first simply breaks each block of text to a sequence of lines, and then breaks each line into a pair of text items:

```
text_parse = lambda text: (r.split() for r in text.splitlines())
```

We've used the `splitlines()` method of the text block to create a sequence of lines. We put this into a generator function so that each individual line could be assigned to `r`. Using `r.split()` separates the two blocks of text in each row.

If we use `list(text_parse(text_1))`, we'll see data like this:

```
[['10', '8.04'],
 ['8', '6.95'],
 ['13', '7.58'],
 ...
 ['5', '5.68']]
```

We need to further enrich this data to make it more usable. We need to convert the strings to proper float values. While doing that, we'll create `SimpleNamespace` instances from each item:

```
from types import SimpleNamespace
row_build = lambda rows: (SimpleNamespace(x=float(x), y=float(y)) for
x,y in rows)
```

The `lambda` object creates a `SimpleNamespace` instance by applying the `float()` function to each string item in each row. This gives us data we can work with.

We can apply these two `lambda` objects to the data to create some usable datasets. Earlier, we showed `text_1`. We'll assume that we have a second, similar set of data assigned to `text_2`:

```
data_1 = list(row_build(text_parse(text_1)))
data_2 = list(row_build(text_parse(text_2)))
```

This creates data from two blocks of similar text. Each has pairs of data points. The `SimpleNamespace` object has two attributes, `x` and `y`, assigned to each row of the data.

Note that this process creates instances of `types.SimpleNamespace`. When we print them, they will be displayed using the class `namespace`. These are mutable objects, so that we can update each one with the standardized z-score.

Printing `data_1` looks like this:

```
    [namespace(x=10.0, y=8.04), namespace(x=8.0, y=6.95),
namespace(x=13.0, y=7.58),
    ...,
    namespace(x=5.0, y=5.68)]
```

As an example, we'll compute a standardized value for the `x` attribute. This means getting mean and standard deviation. Then we'll need to apply these values to standardize data in both of our collections. It looks like this:

```
import statistics
mean_x = statistics.mean(item.x for item in data_1)
stdev_x = statistics.stdev(item.x for item in data_1)

for row in data_1:
    z_x = standardize(mean_x, stdev_x, row.x)
    print(row, z_x)

for row in data_2:
    z_x = standardize(mean_x, stdev_x, row.x)
    print(row, z_x)
```

Providing the `mean_v1`, `stdev_v1` values each time we evaluate `standardize()` can clutter an algorithm with details that aren't deeply important. In some rather complex algorithms, the clutter can lead to more confusion than clarity.

How to do it...

In addition to simply using the `def` statement to create a function that has a partial set of argument values, we have two other ways to create a partial function:

- Using the `partial()` function from the `functools` module
- Creating a `lambda` object

Using functools.partial()

1. Import the `partial` function from `functools`:

```
from functools import partial
```

2. Create an object using `partial()`. We provide the base function, plus the positional arguments that need to be included. Any parameters which are not supplied when the partial is defined must be supplied when the partial is evaluated:

```
z = partial(standardize, mean_x, stdev_x)
```

3. We've provided values for the first two positional parameters, `mean` and `stdev`. The third positional parameter, `x`, must be supplied in order to compute a value.

Creating a lambda object

1. Define a `lambda` object that binds the fixed parameters:

```
lambda x: standardize(mean_v1, stdev_v1, x)
```

2. Create an object using `lambda`:

```
z = lambda x: standardize(mean_v1, stdev_v1, x)
```

How it works...

Both techniques create a callable object - a function - named `z()` that has the values for `mean_v1` and `stdev_v1` already bound to the first two positional parameters. With either approach, we have processing that can look like this:

```
for row in data_1:
    print(row, z(row.x))

for row in data_2:
    print(row, z(row.x))
```

We've applied the `z()` function to each set of data. Because the function has some parameters already applied, its use here looks very simple.

We can also do the following because each row is a mutable object:

```
for row in data_1:
    row.z = z(row.v1)

for row in data_2:
    row.z = z(row.v1)
```

We've updated the row to include a new attribute, `z`, with the value of the `z()` function. In a complex algorithm, tweaking the row objects like this may be a helpful simplification.

There's a significant difference between the two techniques for creating the `z()` function:

- The `partial()` function binds the actual values of the parameters. Any subsequent change to the variables that were used doesn't change the definition of the partial function that's created. After creating `z = partial(standardize(mean_v1, stdev_v1))`, changing the value of `mean_v1` or `stdev_v1` doesn't have an impact on the partial function, `z()`.
- The `lambda` object binds the variable name, not the value. Any subsequent change to the variable's value will change the way the lambda behaves. After creating `z = lambda x: standardize(mean_v1, stdev_v1, x)`, changing the value of `mean_v1` or `stdev_v1` changes the values used by the `lambda` object, `z()`.

We can modify the lambda slightly to bind values instead of names:

```
z = lambda x, m=mean_v1, s=stdev_v1: standardize(m, s, x)
```

This extracts the values of `mean_v1` and `stdev_v1` to create default values for the `lambda` object. The values of `mean_v1` and `stdev_v1` are now irrelevant to proper operation of the `lambda` object, `z()`.

There's more...

We can provide keyword argument values as well as positional argument values when creating a partial function. In many cases, this works nicely. There are a few cases where it doesn't work.

The `reduce()` function, specifically, can't be trivially turned into a partial function. The parameters aren't in the ideal order for creating a partial. The `reduce()` function has the following notional definition. This is not how it's defined - this is how it *appears* to be defined:

```
def reduce(function, iterable, initializer=None)
```

If this was the actual definition, we could do this:

```
prod = partial(reduce(mul, initializer=1))
```

Practically, we can't do this because the definition of `reduce()` is a bit more complex than it might appear. The `reduce()` function doesn't permit named argument values. This means that we're forced to use the lambda technique:

```
>>> from operator import mul
>>> from functools import reduce
>>> prod = lambda x: reduce(mul, x, 1)
```

We've used a `lambda` object to define a function, `prod()`, with only one parameter. This function uses `reduce()` with two fixed parameters, and one variable parameter.

Given this definition for `prod()`, we can define other functions that rely on computing products. Here's a definition of the `factorial` function:

```
>>> factorial = lambda x: prod(range(2,x+1))
>>> factorial(5)
120
```

The definition of `factorial()` depends on `prod()`. The definition of `prod()` is a kind of partial function that uses `reduce()` with two fixed parameter values. We've managed to use a few definitions to create a fairly sophisticated function.

In Python, a function is an object. We've seen numerous ways that functions can be an argument to a function. A function that accepts another function as an argument is sometimes called a **higher-order function**.

Similarly, functions can also return a function object as a result. This means that we can create a function like this:

```
def prepare_z(data):
    mean_x = statistics.mean(item.x for item in data_1)
    stdev_x = statistics.stdev(item.x for item in data_1)
    return partial(standardize, mean_x, stdev_x)
```

We've defined a function over a set of (x,y) samples. We've computed the mean and standard deviation of the x attribute of each sample. We've then created a partial function which can standardize scores based on the computed statistics. The result of this function is a function we can use for data analysis:

```
z = prepare_z(data_1)
for row in data_2:
    print(row, z(row.x))
```

When we evaluated the `prepare_z()` function, it returned a function. We assigned this function to a variable, `z`. This variable is a callable object; it's the function `z()` that will standardize a score based on the sample mean and standard deviation.

Simplifying complex algorithms with immutable data structures

The concept of a stateful object is a common feature of object-oriented programming. We looked at a number of techniques related to objects and state in Chapter 6, *Basics of Classes and Objects*, and Chapter 7, *More Advanced Class Design*. A great deal of the emphasis of object-oriented design is creating methods that mutate an object's state.

We've also looked at some stateful functional programming techniques in the *Using stacked generator expressions*, *Combining map and reduce transformations*, and *Creating a partial function* recipes. We've used `types.SimpleNamespace` because it creates a simple, stateful object with easy to use attribute names.

In most of these cases, we've been working with objects that have a Python `dict` object that defines the attributes. The one exception is the *Optimizing small objects with __slots__* recipe, where the attributes are fixed by the `__slots__` attribute definition.

Using a `dict` object to store an object's attributes has several consequences:

- We can trivially add and remove attributes. We're not limited to simply setting and getting defined attributes; we can create new attributes too.
- Each object uses a somewhat larger amount of memory than is minimally necessary. This is because dictionaries use a hashing algorithm to locate keys and values. The hash processing generally requires more memory than other structures such as a `list` or a `tuple`. For very large amounts of data, this can become a problem.

The most significant issue with stateful object-oriented programming is that it can sometimes be challenging to write clear assertions about state change of an object. Rather than defining assertions about state change, it's much easier to create entirely new objects with a state that can be simply mapped to the object's type. This, coupled with Python type hints, can sometimes create more reliable, and easier to test, software.

When we create new objects, the relationships between data items and computations can be captured explicitly. The `mypy` project provides tools that can analyze those type hints to provide some confirmation that the objects used in a complex algorithm are used properly.

In some cases, we can also reduce the amount of memory by avoiding stateful objects in the first place. We have two techniques for doing this:

- Using class definitions with `__slots__`: See the *Optimizing small objects with __slots__* recipe for this. These objects are mutable, so we can update attributes with new values.
- Using immutable `tuples` or `namedtuples`: See the *Designing classes with little unique processing* recipe for some background on this. These objects are immutable. We can create new objects, but we can't change the state of an object. The cost savings from less memory overall have to be balanced against the additional costs of creating new objects.

Immutable objects can be somewhat faster that mutable objects. The more important benefit is to algorithm design. In some cases, writing functions that create new immutable objects from old immutable objects can be simpler, and easier to test and debug, than algorithms that work with stateful objects. Writing type hints can help this process.

Getting ready

As we noted in the *Using stacked generator expressions* and *Implementing "there exists" processing* recipes, we can only process a generator once. If we need to process it more than one time, the iterable sequence of objects must be transformed into a complete collection like a list or tuple.

This often leads to a multi-phase process:

- **Initial extract of the data**: This might involve a database query, or reading a `.csv` file. This phase can be implemented as a function that yields rows or even returns a generator function.
- **Cleansing and filtering the data**: This may involve a stack of generator expressions that can process the source just once. This phase is often implemented as a function that includes several map and filter operations.
- **Enriching the data**: This, too, may involve a stack of generator expressions that can process the data one row at a time. This is typically a series of map operations to create new, derived data from existing data.
- **Reducing or summarizing the data**: This may involve multiple summaries. In order for this to work, the output from the enrichment phase needs to be a collection object that can be processed more than one time.

In some cases, the enrichment and summary processes may be interleaved. As we saw in the *Creating a partial function* recipe, we might do some summarization followed by more enrichment.

There are two common strategies for handling the enriching phase:

- **Mutable objects**: This means that enrichment processing adds or sets values of attributes. This can be done with eager calculations as attributes are set. See the *Using settable properties to update eager attributes* recipe. It can also be done with lazy properties. See the *Using properties for lazy attributes* recipe. We've shown examples using `types.SimpleNamespace` where the computation is done in functions separate from the class definition.
- **Immutable objects**: This means that the enrichment process creates new objects from old objects. Immutable objects are derived from `tuple` or are a type created by `namedtuple()`. These objects have the advantage of being very small and very fast. Also, the lack of any internal state change can make them very simple.

Let's say we've got a collection of data samples in big blocks of text:

```
text_1 = '''10  8.04
8       6.95
13      7.58
...
5       5.68
'''
```

Our goal is a three-step process that includes the `get`, `cleanse`, and `enrich` operations:

```
data = list(enrich(cleanse(get(text))))
```

The `get()` function acquires the data from a source; in this case, it would parse the big block of text. The `cleanse()` function would remove blank lines and other unusable data. The `enrich()` function would do the final calculation on the cleaned data. We'll look at each phase of this pipeline separately.

The `get()` function is limited to pure text processing, doing as little filtering as possible:

```
from typing import *

def get(text: str) -> Iterator[List[str]]:
    for line in text.splitlines():
        if len(line) == 0:
            continue
        yield line.split()
```

In order to write type hints, we've imported the `typing` module. This allows us to make an explicit declaration about the inputs and outputs of this function. The `get()` function accepts a string, `str`. It yields a `List[str]` structure. Each line of input is decomposed to a sequence of values.

This function will generate all non-empty lines of data. There is a small filtering feature to this, but it's related to a small technical issue around data serialization, not an application-specific filtering rule.

The `cleanse()` function will generate named tuples of data. This will apply a number of rules to assure that the data is valid:

```
from collections import namedtuple

DataPair = namedtuple('DataPair', ['x', 'y'])

def cleanse(iterable: Iterable[List[str]]) -> Iterator[DataPair]:
    for text_items in iterable:
        try:
            x_amount = float(text_items[0])
            y_amount = float(text_items[1])
            yield DataPair(x_amount, y_amount)
        except Exception as ex:
            print(ex, repr(text_items))
```

We've defined a `namedtuple` with the uninformative name of `DataPair`. This item has two attributes, `x`, and `y`. If the two text values can be properly converted to `float`, then this generator will yield a useful `DataPair` instance. If the two text values cannot be converted, this will display an error for the offending pair.

Note the technical subtlety that's part of the `mypy` project's type hints. A function with a `yield` statement is an iterator. We can use it as if it's an iterable object because of a formal relationship that says iterators are a kind of iterable.

Additional cleansing rules could be applied here. For example, `assert` statements could be added inside the `try` statement. Any exception raised by unexpected or invalid data will stop processing the given row of input.

Here's the result of this initial `cleanse()` and `get()` processing:

```
list(cleanse(get(text)))
The output looks like this:
[DataPair(x=10.0, y=8.04),
 DataPair(x=8.0, y=6.95),
 DataPair(x=13.0, y=7.58),
 ...,
 DataPair(x=5.0, y=5.68)]
```

In this example, we'll rank order by the `y` value of each pair. This requires sorting the data first, and then yielding the sorted values with an additional attribute value, the `y` rank order.

How to do it...

1. Define the enriched `namedtuple`:

   ```
   RankYDataPair = namedtuple('RankYDataPair', ['y_rank', 'pair'])
   ```

   Note that we've specifically included the original pair as a data item in this new data structure. We don't want to copy the individual fields; instead, we've incorporated the original object as a whole.

2. Define the enrichment function:

   ```
   PairIter = Iterable[DataPair]
   RankPairIter = Iterator[RankYDataPair]

   def rank_by_y(iterable:PairIter) -> RankPairIter:
   ```

We've included type hints on this function to make it clear precisely what types are expected and returned by this enrichment function. We defined the type hints separately so that they're shorter and so that they can be reused in other functions.

3. Write the body of the enrichment. In this case, we're going to be rank ordering, so we'll need sorted data, using the original `y` attribute. We're creating new objects from the old objects, so the function yields instances of `RankYDataPair`:

```
all_data = sorted(iterable, key=lambda pair:pair.y)
for y_rank, pair in enumerate(all_data, start=1):
    yield RankYDataPair(y_rank, pair)
```

We've used `enumerate()` to create the rank order numbers to each value. The starting value of `1` is sometimes handy for some statistical processing. In other cases, the default starting value of `0` will work out well.

Here's the whole function:

```
def rank_by_y(iterable: PairIter) -> RankPairIter:
    all_data = sorted(iterable, key=lambda pair:pair.y)
    for y_rank, pair in enumerate(all_data, start=1):
        yield RankYDataPair(y_rank, pair)
```

We can use this in a longer expression to get, cleanse, and then rank. The use of type hints can make this clearer than an alternative involving stateful objects. In some cases, there can be a very helpful improvement in the clarity of the code.

How it works...

The result of the `rank_by_y()` function is a new object which contains the original object, plus the result of the enrichment. Here's how we'd use this stacked sequence of generators: `rank_by_y()`, `cleanse()`, and `get()`:

```
>>> data = rank_by_y(cleanse(get(text_1)))
>>> pprint(list(data))
[RankYDataPair(y_rank=1, pair=DataPair(x=4.0, y=4.26)),
 RankYDataPair(y_rank=2, pair=DataPair(x=7.0, y=4.82)),
 RankYDataPair(y_rank=3, pair=DataPair(x=5.0, y=5.68)),
 ...,
 RankYDataPair(y_rank=11, pair=DataPair(x=12.0, y=10.84))]
```

The data is in ascending order by the `y` value. We can now use these enriched data values for further analysis and calculation.

Creating new objects can - in many cases - be more expressive of the algorithm than changing the state of objects. This is often a subjective judgement.

The Python type hints work best with the creation of new objects. Consequently, this technique can provide strong evidence that a complex algorithm is correct. Using `mypy` makes immutable objects more appealing.

Finally, we may sometimes see a small speed-up when we use immutable objects. This relies on a balance between three features of Python to be effective:

- Tuples are small data structures. Using these can improve performance.
- Any relationship between objects in Python involves creating an object reference, a data structure that's also very small. A number of related immutable objects might be smaller than a mutable object.
- Object creation can be costly. Creating too many immutable objects outweighs the benefits.

The memory savings from the first two features must be balanced against the processing cost from the third feature. Memory savings can lead to better performance when there's a huge volume of data that constrains processing speeds.

For small examples like this one, the volume of data is so tiny that the object creation cost is large compared with any cost savings from reducing the volume of memory in use. For larger sets of data, the object creation cost may be less than the cost of running low on memory.

There's more...

The `get()` and `cleanse()` functions in this recipe both refer to a similar data structure: `Iterable[List[str]]` and `Iterator[List[str]]`. In the `collections.abc` module, we see that `Iterable` is the generic definition, and `Iterator` is a special case of `Iterable`.

The `mypy` release used for this book—`mypy 0.2.0-dev`—is very particular about functions with the `yield` statement being defined as an `Iterator`. A future release may relax this strict check of the subclass relationship, allowing us to use one definition for both cases.

The `typing` module includes an alternative to the `namedtuple()` function: `NamedTuple()`. This allows specification of a data type for the various items within the tuple.

It looks like this:

```
DataPair = NamedTuple('DataPair', [
        ('x', float),
        ('y', float)
    ]
)
```

We use `typing.NamedTuple()` almost exactly the same way we use `collection.namedtuple()`. The definition of the attributes uses a list of two-tuples instead of a list of names. The two-tuples have a name and a type definition.

This supplemental type definition is used by `mypy` to determine whether the `NamedTuple` objects are being populated correctly. It can also be used by other people to understand the code and make proper modifications or extensions.

In Python, we can replace some stateful objects with immutable objects. There are a number of limitations, though. The collections such as list, set, and dict, must remain as mutable objects. Replacing these collections with immutable monads can work out well in other programming languages, but it's not a part of Python.

Writing recursive generator functions with the yield from statement

There are a number of algorithms that can be expressed neatly as recursions. In the *Designing recursive functions around Python's stack limits* recipe, we looked at some recursive functions that could be optimized to reduce the number of function calls.

When we look at some data structures, we see that they involve recursion. In particular, JSON documents (as well as XML and HTML documents) can have a recursive structure. A JSON document might include a complex object that contains other complex objects within it.

In many cases, there are advantages to using generators for processing these kinds of structures. How can we write generators that work with recursion? How does the `yield from` statement save us from writing an extra loop?

Getting ready

We'll look at a way to search an ordered collection for all matching values in a complex data structure. When working with complex JSON documents, we'll often model them as dict-of-dict, and dict-of-list structures. Of course, a JSON document is not a two-level thing; dict-of-dict really means dict-of-dict-of.... Similarly, dict-of-list really means dict-of-list-of... These are recursive structures, which means a search must descend through the entire structure looking for a particular key or value.

A document with this complex structure might look like this:

```
document = {
    "field": "value1",
    "field2": "value",
    "array": [
        {"array_item_key1": "value"},
        {"array_item_key2": "array_item_value2"}
    ],
    "object": {
        "attribute1": "value",
        "attribute2": "value2"
    }
}
```

This shows a document that has four keys, `field`, `field2`, `array`, and `object`. Each of these keys has a distinct data structure as its associated value. Some of the values are unique, and some are duplicated. This duplication is the reason why our search must find **all** instances inside the overall document.

The core algorithm is a depth-first search. The output from this function will be a list of paths that identify the target value. Each path will be a sequence of field names or field names mixed with index positions.

In the previous example, the value `value` can be found in three places:

- `["array", 0, "array_item_key1"]`: This path starts with the top-level field named `array`, then visits item zero of a list, then a field named `array_item_key1`
- `["field2"]`: This path has just a single field name where the value is found
- `["object", "attribute1"]`: This path starts with the top-level field named `object`, then the child `attribute1` of that field

The `find_value()` function yield both of these paths when it searches the overall document for the target value. Here's the conceptual overview of this search function:

```
def find_path(value, node, path=[]):
    if isinstance(node, dict):
        for key in node.keys():
            # find_value(value, node[key], path+[key])
            # This must yield multiple values
    elif isinstance(node, list):
        for index in range(len(node)):
            # find_value(value, node[index], path+[index])
            # This will yield multiple values
    else:
        # a primitive type
        if node == value:
            yield path
```

There are three alternatives in the `find_path()` process:

- When the node is a dictionary, the value of each key must be examined. The values may be any kind of data, so we'll use the `find_path()` function recursively on each value. This will yield a sequence of matches.
- If node is a list, the items for each index position must be examined. The items may be any kind of data, so we'll use the `find_path()` function recursively on each value. This will yield a sequence of matches.
- The other choice is for the node to be a primitive value. The JSON specification lists a number of primitives that may be present in a valid document. If the node value is the target value, we've found one instance, and can yield this single match.

There are two ways to handle the recursion. One is like this:

```
for match in find_value(value, node[key], path+[key]):
    yield match
```

This seems to have too much boilerplate for such a simple idea. The other way is simpler and a bit clearer.

How to do it...

1. Write out the complete `for` statement:

```
for match in find_value(value, node[key], path+[key]):
    yield match
```

   For debugging purposes, we might insert a `print()` function inside the body of the `for` statement.

2. Replace this with a `yield from` statement once we're sure things work:

```
yield from find_value(value, node[key], path+[key])
```

The complete depth-first `find_value()` search function will look like this:

```
def find_path(value, node, path=[]):
    if isinstance(node, dict):
        for key in node.keys():
            yield from find_path(value, node[key], path+[key])
    elif isinstance(node, list):
        for index in range(len(node)):
            yield from find_path(value, node[index], path+[index])
    else:
        if node == value:
            yield path
```

When we use the `find_path()` function, it looks like this:

```
>>> list(find_path('array_item_value2', document))
[['array', 1, 'array_item_key2']]
```

The `find_path()` function is iterable. It can yield a number of values. We consumed all of the results to create a list. In this example, the list had one item, `['array', 1, 'array_item_key2']`. This item has the path to the matching item.

We can then evaluate `document['array'][1]['array_item_key2']` to find the referenced value.

When we look for a non-unique value, we might see a list like this:

```
>>> list(find_value('value', document))
[['array', 0, 'array_item_key1'],
 ['field2'],
 ['object', 'attribute1']]
```

The resulting list has three items. Each of these provides the path to an item with the target value of `value`.

How it works...

The `yield from X` statement is shorthand for:

```
for item in X:
    yield item
```

This lets us write a succinct recursive algorithm that will behave as an iterator and properly yield multiple values.

This can also be used in contexts that don't involve a recursive function. It's entirely sensible to use a `yield from` statement anywhere that an iterable result is involved. It's a big simplification for recursive functions, however, because it preserves a clearly recursive structure.

There's more...

Another common style of definition assembles a list using append operations. We can rewrite this into an iterator and avoid the overhead of building a list object.

When factoring a number, we can define the set of prime factors like this:

$$F(x) = \begin{cases} \{x\} & \text{if } x \text{ is prime} \\ \{n\} \cup F\left(\frac{x}{n}\right) & \text{if } n \text{ is a factor of x} \end{cases}$$

If the value, x, is prime, it has only itself in the set of prime factors. Otherwise, there must be some prime number, n, which is the least factor of x. We can assemble a set of factors starting with n and including all factors of x/n. In order to be sure that only prime factors are found, then n must be prime. If we search in ascending order, we'll find prime factors before finding composite factors.

We have two ways to implement this in Python: one builds a list, the other generates factors. Here's a list-building function:

```
import math
def factor_list(x):
    limit = int(math.sqrt(x)+1)
    for n in range(2, limit):
        q, r = divmod(x, n)
        if r == 0:
            return [n] + factor_list(q)
    return [x]
```

This `factor_list()` function will search all numbers, n, such that $2 \le n < \sqrt{x}$. The first number that's a factor of x will be the least factor. It will also be prime. We'll—of course—search a number of composite values, wasting time. For example, after testing two and three, we'll also test values for line four and six, even though they're composite and all of their factors have already been tested.

This function builds a `list` object. If a factor, `n`, is found, it will start a list with that factor. It will append factors from `x // n`. If there are no factors of `x`, then the value is prime, and we return a list with just that value.

We can rewrite this to be an iterator by replacing the recursive calls with `yield from`. The function will look like this:

```
def factor_iter(x):
    limit = int(math.sqrt(x)+1)
    for n in range(2, limit):
        q, r = divmod(x, n)
        if r == 0:
            yield n
            yield from factor_iter(q)
            return
    yield x
```

As with the list-building version, this will search numbers, n, such that . When a factor is found, then the function will yield the factor, followed by any other factors found by a recursive call to `factor_iter()`. If no factors are found, the function will yield just the prime number and nothing more.

Using an iterator allows us to build any kind of collection from the factors. Instead of being limited to always creating a *list*, we can create a multiset using the `collection.Counter` class. It would look like this:

```
>>> from collections import Counter
>>> Counter(factor_iter(384))
Counter({2: 7, 3: 1})
```

This shows us that:

$$384 = 2^7 \times 3$$

In some cases, this kind of multiset is easier to work with than the list of factors.

See also

- In the *Designing recursive functions around Python's stack limits* recipe, we cover the core design patterns for recursive functions. This recipe provides an alternative way to create the results.