Chapter 5: Sequences and Coroutines

Contents

5.1 Introduction

In this chapter, we continue our discussion of real-world applications by developing new tools to process sequential data. In Chapter 2, we introduced a sequence interface, implemented in Python by built-in data types such as `tuple` and `list`. Sequences supported two operations: querying their length and accessing an element by index. In Chapter 3, we developed a user-defined implementations of the sequence interface, the `Rlist` class for representing recursive lists. These sequence types proved effective for representing and accessing a wide variety of sequential datasets.

However, representing sequential data using the sequence abstraction has two important limitations. The first is that a sequence of length n typically takes up an amount of memory proportional to n. Therefore, the longer a sequence is, the more memory it takes to represent it.

The second limitation of sequences is that sequences can only represent datasets of known, finite length. Many sequential collections that we may want to represent do not have a well-defined length, and some are even infinite. Two mathematical examples of infinite sequences are the positive integers and the Fibonacci numbers. Sequential data sets of unbounded length also appear in other computational domains. For instance, the sequence of all Twitter posts grows longer with every second and therefore does not have a fixed length. Likewise, the sequence of telephone calls sent through a cell tower, the sequence of mouse movements made by a computer user, and the sequence of acceleration measurements from sensors on an aircraft all extend without bound as the world evolves.

In this chapter, we introduce new constructs for working with sequential data that are designed to accommodate collections of unknown or unbounded length, while using limited memory. We also discuss how these tools can be used with a programming construct called a coroutine to create efficient, modular data processing pipelines.

5.2 Implicit Sequences

The central observation that will lead us to efficient processing of sequential data is that a sequence can be *represented* using programming constructs without each element being *stored* explicitly in the memory of the computer. To put this idea into practice, we will construct objects that provides access to all of the elements of some sequential dataset that an application may desire, but without computing all of those elements in advance and storing them.

A simple example of this idea arises in the `range` sequence type introduced in Chapter 2. A `range` represents a consecutive, bounded sequence of integers. However, it is not the case that each element of that sequence is represented explicitly in memory. Instead, when an element is requested from a `range`, it is computed. Hence, we can represent very large ranges of integers without using large blocks of memory. Only the end points of the range are stored as part of the `range` object, and elements are computed on the fly.

```
>>> r = range(10000, 1000000000)
>>> r[45006230]
45016230
```

In this example, not all 999,990,000 integers in this range are stored when the range instance is constructed. Instead, the range object adds the first element 10,000 to the index 45,006,230 to produce the element 45,016,230. Computing values on demand, rather than retrieving them from an existing representation, is an example of *lazy* computation. Computer science is a discipline that celebrates laziness as an important computational tool.

An *iterator* is an object that provides sequential access to an underlying sequential dataset. Iterators are built-in objects in many programming languages, including Python. The iterator abstraction has two components: a mechanism for retrieving the *next* element in some underlying series of elements and a mechanism for signaling that the end of the series has been reached and no further elements remain. In programming languages with built-in object systems, this abstraction typically corresponds to a particular interface that can be implemented by classes. The Python interface for iterators is described in the next section.

The usefulness of iterators is derived from the fact that the underlying series of data for an iterator may not be represented explicitly in memory. An iterator provides a mechanism for considering each of a series of values in turn, but all of those elements do not need to be stored simultaneously. Instead, when the next element is requested from an iterator, that element may be computed on demand instead of being retrieved from an existing memory source.

Ranges are able to compute the elements of a sequence lazily because the sequence represented is uniform, and any element is easy to compute from the starting and ending bounds of the range. Iterators allow for lazy generation of a much broader class of underlying sequential datasets, because they do not need to provide access to arbitrary elements of the underlying series. Instead, they must only compute the next element of the series, in order, each time another element is requested. While not as flexible as accessing arbitrary elements of a sequence (called *random access*), *sequential access* to sequential data series is often sufficient for data processing applications.

5.2.1 Python Iterators

The Python iterator interface includes two messages. The `__next__` message queries the iterator for the next element of the underlying series that it represents. In response to invoking `__next__` as a method, an iterator can perform arbitrary computation in order to either retrieve or compute the next element in an underlying series. Calls to `__next__` make a mutating change to the iterator: they advance the position of the iterator. Hence, multiple calls to `__next__` will return sequential elements of an underlying series. Python signals that the end of an underlying series has been reached by raising a `StopIteration` exception during a call to `__next__`.

The `Letters` class below iterates over an underlying series of letters from `a` to `d`. The member variable `current` stores the current letter in the series, and the `__next__` method returns this letter and uses it to compute a new value for `current`.

```
>>> class Letters(object):
        def __init__(self):
            self.current = 'a'
        def __next__(self):
            if self.current > 'd':
                raise StopIteration
            result = self.current
            self.current = chr(ord(result)+1)
            return result
        def __iter__(self):
            return self
```

The `__iter__` message is the second required message of the Python iterator interface. It simply returns the iterator; it is useful for providing a common interface to iterators and sequences, as described in the next section.

Using this class, we can access letters in sequence.

```
>>> letters = Letters()
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

A `Letters` instance can only be iterated through once. Once its `__next__()` method raises a `StopIteration` exception, it continues to do so from then on. There is no way to reset it; one must create a new instance.

Iterators also allow us to represent infinite series by implementing a `__next__` method that never raises a `StopIteration` exception. For example, the `Positives` class below iterates over the infinite series of positive integers.

```
>>> class Positives(object):
        def __init__(self):
            self.current = 0;
        def __next__(self):
            result = self.current
            self.current += 1
            return result
        def __iter__(self):
            return self
```

5.2.2 For Statements

In Python, sequences can expose themselves to iteration by implementing the `__iter__` message. If an object represents sequential data, it can serve as an *iterable* object in a `for` statement by returning an iterator object in response to the `__iter__` message. This iterator is meant to have a `__next__()` method that returns each element of the sequence in turn, eventually raising a `StopIteration` exception when the end of the sequence is reached.

```
>>> counts = [1, 2, 3]
>>> for item in counts:
        print(item)
1
2
3
```

In the above example, the `counts` list returns an iterator in response to a call to its `__iter__()` method. The `for` statement then calls that iterator's `__next__()` method repeatedly, and assigns the returned value to `item` each time. This process continues until the iterator raises a `StopIteration` exception, at which point the `for` statement concludes.

With our knowledge of iterators, we can implement the evaluation rule of a `for` statement in terms of `while`, assignment, and `try` statements.

```
>>> i = counts.__iter__()
>>> try:
        while True:
            item = i.__next__()
            print(item)
    except StopIteration:
        pass
1
2
3
```

Above, the iterator returned by invoking the `__iter__` method of `counts` is bound to a name `i` so that it can be queried for each element in turn. The handling clause for the `StopIteration` exception does nothing, but handling the exception provides a control mechanism for exiting the `while` loop.

5.2.3 Generators and Yield Statements

The `Letters` and `Positives` objects above require us to introduce a new field `self.current` into our object to keep track of progress through the sequence. With simple sequences like those shown above, this can be done easily. With complex sequences, however, it can be quite difficult for the `__next__()` function to save its place in the calculation. Generators allow us to define more complicated iterations by leveraging the features of the Python interpreter.

A *generator* is an iterator returned by a special class of function called a *generator function*. Generator functions are distinguished from regular functions in that rather than containing `return` statements in their body, they use `yield` statement to return elements of a series.

Generators do not use attributes of an object to track their progress through a series. Instead, they control the execution of the generator function, which runs until the next `yield` statement is executed each time the generator's `__next__` method is invoked. The `Letters` iterator can be implemented much more compactly using a generator function.

```
>>> def letters_generator():
        current = 'a'
        while current <= 'd':
            yield current
            current = chr(ord(current)+1)

>>> for letter in letters_generator():
        print(letter)
a
b
c
d
```

Even though we never explicitly defined `__iter__()` or `__next__()` methods, Python understands that when we use the `yield` statement, we are defining a generator function. When called, a generator function doesn't return a particular yielded value, but instead a `generator` (which is a type of iterator) that itself can return the yielded values. A generator object has `__iter__` and `__next__` methods, and each call to `__next__` continues execution of the generator function from wherever it left off previously until another `yield` statement is executed.

The first time `__next__` is called, the program executes statements from the body of the `letters_generator` function until it encounters the `yield` statement. Then, it pauses and returns the value of `current`. `yield` statements do not destroy the newly created environment, they preserve it for later. When `__next__` is called again, execution resumes where it left off. The values of `current` and of any other bound names in the scope of `letters_generator` are preserved across subsequent calls to `__next__`.

We can walk through the generator by manually calling `__next__()`:

```
>>> letters = letters_generator()
>>> type(letters)
<class 'generator'>
>>> letters.__next__()
'a'
>>> letters.__next__()
'b'
>>> letters.__next__()
'c'
>>> letters.__next__()
'd'
>>> letters.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The generator does not start executing any of the body statements of its generator function until the first time `__next__()` is called.

5.2.4 Iterables

In Python, iterators only make a single pass over the elements of an underlying series. After that pass, the iterator will continue to raise a `StopIteration` exception when `__next__()` is called. Many applications require iteration over elements multiple times. For example, we have to iterate over a list many times in order to enumerate all pairs of elements.

```
>>> def all_pairs(s):
        for item1 in s:
            for item2 in s:
                yield (item1, item2)

>>> list(all_pairs([1, 2, 3]))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Sequences are not themselves iterators, but instead *iterable* objects. The iterable interface in Python consists of a single message, `__iter__`, that returns an iterator. The built-in sequence types in Python return new instances of iterators when their `__iter__` methods are invoked. If an iterable object returns a fresh instance of an iterator each time `__iter__` is called, then it can be iterated over multiple times.

New iterable classes can be defined by implementing the iterable interface. For example, the *iterable* `LetterIterable` class below returns a new iterator over letters each time `__iter__` is invoked.

```
>>> class LetterIterable(object):
        def __iter__(self):
            current = 'a'
            while current <= 'd':
                yield current
                current = chr(ord(current)+1)
```

The `__iter__` method is a generator function; it returns a generator object that yields the letters `'a'` through `'d'`.

A `Letters` iterator object gets "used up" after a single iteration, whereas the `LetterIterable` object can be iterated over multiple times. As a result, a `LetterIterable` instance can serve as an argument to `all_pairs`.

```
>>> letters = LetterIterable()
>>> all_pairs(letters).__next__()
('a', 'a')
```

5.2.5 Streams

Streams offer a final way to represent sequential data implicitly. A stream is a lazily computed recursive list. Like the `Rlist` class from Chapter 3, a `Stream` instance responds to requests for its `first` element and the `rest` of the stream. Like an `Rlist`, the `rest` of a `Stream` is itself a `Stream`. Unlike an `Rlist`, the `rest` of a stream is only computed when it is looked up, rather than being stored in advance. That is, the `rest` of a stream is computed lazily.

To achieve this lazy evaluation, a stream stores a function that computes the rest of the stream. Whenever this function is called, its returned value is cached as part of the stream in an attribute called `_rest`, named with an underscore to indicate that it should not be accessed directly. The accessible attribute `rest` is a property method that returns the rest of the stream, computing it if necessary. With this design, a stream stores *how to compute* the rest of the stream, rather than always storing it explicitly.

```
>>> class Stream(object):
        """A lazily computed recursive list."""
        def __init__(self, first, compute_rest, empty=False):
            self.first = first
            self._compute_rest = compute_rest
            self.empty = empty
            self._rest = None
```

```
            self._computed = False
        @property
        def rest(self):
            """Return the rest of the stream, computing it if necessary."""
            assert not self.empty, 'Empty streams have no rest.'
            if not self._computed:
                self._rest = self._compute_rest()
                self._computed = True
            return self._rest
        def __repr__(self):
            if self.empty:
                return '<empty stream>'
            return 'Stream({0}, <compute_rest>)'.format(repr(self.first))
>>> Stream.empty = Stream(None, None, True)
```

A recursive list is defined using a nested expression. For example, we can create an `Rlist` that represents the elements `1` then `5` as follows:

```
>>> r = Rlist(1, Rlist(2+3, Rlist.empty))
```

Likewise, we can create a `Stream` representing the same series. The `Stream` does not actually compute the second element `5` until the rest of the stream is requested.

```
>>> s = Stream(1, lambda: Stream(2+3, lambda: Stream.empty))
```

Here, `1` is the first element of the stream, and the `lambda` expression that follows returns a function for computing the rest of the stream. The second element of the computed stream is a function that returns an empty stream.

Accessing the elements of recursive list `r` and stream `s` proceed similarly. However, while `5` is stored within `r`, it is computed on demand for `s` via addition the first time that it is requested.

```
>>> r.first
1
>>> s.first
1
>>> r.rest.first
5
>>> s.rest.first
5
>>> r.rest
Rlist(5)
>>> s.rest
Stream(5, <compute_rest>)
```

While the `rest` of `r` is a one-element recursive list, the `rest` of `s` includes a function to compute the rest; the fact that it will return the empty stream may not yet have been discovered.

When a `Stream` instance is constructed, the field `self._computed` is `False`, signifying that the `_rest` of the `Stream` has not yet been computed. When the `rest` attribute is requested via a dot expression, the `rest` method is invoked, which triggers computation with `self._rest = self.compute_rest`. Because of the caching mechanism within a `Stream`, the `compute_rest` function is only ever called once.

The essential properties of a `compute_rest` function are that it takes no arguments, and it returns a `Stream`.

Lazy evaluation gives us the ability to represent infinite sequential datasets using streams. For example, we can represent increasing integers, starting at any `first` value.

```
>>> def make_integer_stream(first=1):
      def compute_rest():
        return make_integer_stream(first+1)
      return Stream(first, compute_rest)
```

```
>>> ints = make_integer_stream()
>>> ints
Stream(1, <compute_rest>)
>>> ints.first
1
```

When `make_integer_stream` is called for the first time, it returns a stream whose `first` is the first integer in the sequence (`1` by default). However, `make_integer_stream` is actually recursive because this stream's `compute_rest` calls `make_integer_stream` again, with an incremented argument. This makes `make_integer_stream` recursive, but also lazy.

```
>>> ints.first
1
>>> ints.rest.first
2
>>> ints.rest.rest
Stream(3, <compute_rest>)
```

Recursive calls are only made to `make_integer_stream` whenever the `rest` of an integer stream is requested.

The same higher-order functions that manipulate sequences -- `map` and `filter` -- also apply to streams, although their implementations must change to apply their argument functions lazily. The function `map_stream` maps a function over a stream, which produces a new stream. The locally defined `compute_rest` function ensures that the function will be mapped onto the rest of the stream whenever the rest is computed.

```
>>> def map_stream(fn, s):
        if s.empty:
            return s
        def compute_rest():
            return map_stream(fn, s.rest)
        return Stream(fn(s.first), compute_rest)
```

A stream can be filtered by defining a `compute_rest` function that applies the filter function to the rest of the stream. If the filter function rejects the first element of the stream, the rest is computed immediately. Because `filter_stream` is recursive, the rest may be computed multiple times until a valid `first` element is found.

```
>>> def filter_stream(fn, s):
        if s.empty:
            return s
        def compute_rest():
            return filter_stream(fn, s.rest)
        if fn(s.first):
            return Stream(s.first, compute_rest)
        return compute_rest()
```

The `map_stream` and `filter_stream` functions exhibit a common pattern in stream processing: a locally defined `compute_rest` function recursively applies a processing function to the rest of the stream whenever the rest is computed.

To inspect the contents of a stream, we can truncate it to finite length and convert it to a Python `list`.

```
>>> def truncate_stream(s, k):
        if s.empty or k == 0:
            return Stream.empty
        def compute_rest():
            return truncate_stream(s.rest, k-1)
        return Stream(s.first, compute_rest)

>>> def stream_to_list(s):
        r = []
```

```
        while not s.empty:
            r.append(s.first)
            s = s.rest
        return r
```

These convenience functions allow us to verify our `map_stream` implementation with a simple example that squares the integers from `3` to `7`.

```
>>> s = make_integer_stream(3)
>>> s
Stream(3, <compute_rest>)
>>> m = map_stream(lambda x: x*x, s)
>>> m
Stream(9, <compute_rest>)
>>> stream_to_list(truncate_stream(m, 5))
[9, 16, 25, 36, 49]
```

We can use our `filter_stream` function to define a stream of prime numbers using the sieve of Eratosthenes, which filters a stream of integers to remove all numbers that are multiples of its first element. By successively filtering with each prime, all composite numbers are removed from the stream.

```
>>> def primes(pos_stream):
        def not_divible(x):
            return x % pos_stream.first != 0
        def compute_rest():
            return primes(filter_stream(not_divible, pos_stream.rest))
        return Stream(pos_stream.first, compute_rest)
```

By truncating the `primes` stream, we can enumerate any prefix of the prime numbers.

```
>>> p1 = primes(make_integer_stream(2))
>>> stream_to_list(truncate_stream(p1, 7))
[2, 3, 5, 7, 11, 13, 17]
```

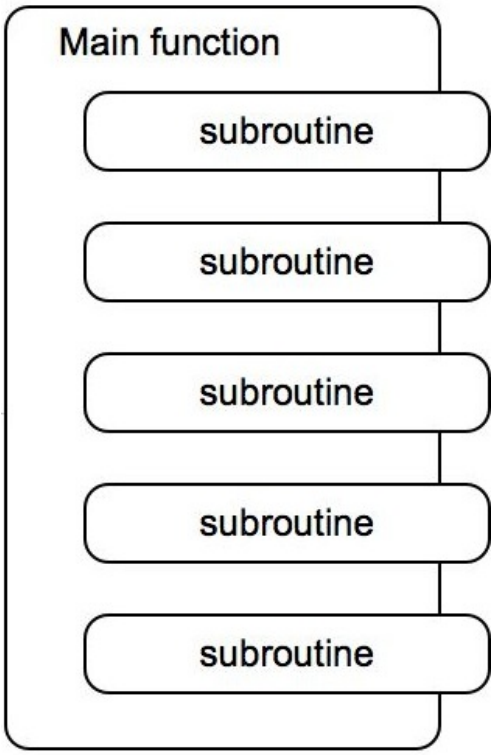
Streams contrast with iterators in that they can be passed to pure functions multiple times and yield the same result each time. The primes stream is not "used up" by converting it to a list. That is, the `first` element of `p1` is still `2` after converting the prefix of the stream to a list.

```
>>> p1.first
2
```

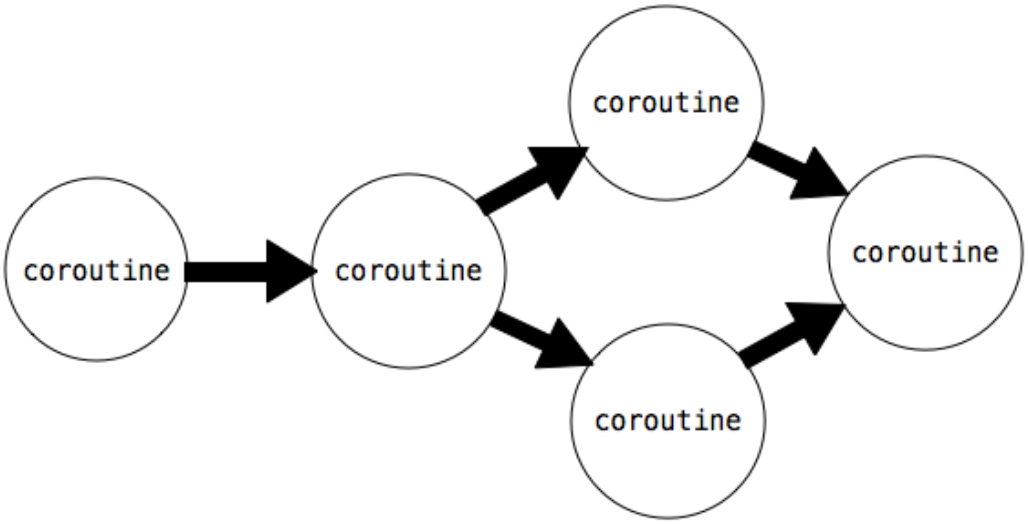
Just as recursive lists provide a simple implementation of the sequence abstraction, streams provide a simple, functional, recursive data structure that implements lazy evaluation through the use of higher-order functions.

5.3 Coroutines

Much of this text has focused on techniques for decomposing complex programs into small, modular components. When the logic for a function with complex behavior is divided into several self-contained steps that are themselves functions, these functions are called helper functions or *subroutines*. Subroutines are called by a main function that is responsible for coordinating the use of several subroutines.



In this section, we introduce a different way of decomposing complex computations using *coroutines*, an approach that is particularly applicable to the task of processing sequential data. Like a subroutine, a coroutine computes a single step of a complex computation. However, when using coroutines, there is no main function to coordinate results. Instead coroutines themselves link together to form a pipeline. There may be a coroutine for consuming the incoming data and sending it to other coroutines. There may be coroutines that each do simple processing steps on data sent to them, and there may finally be another coroutine that outputs a final result.



The difference between coroutines and subroutines is conceptual: subroutines slot into an overarching function to which they are subordinate, whereas coroutines are all colleagues, they cooperate to form a pipeline without any supervising function responsible for calling them in a particular order.

In this section, we will learn how Python supports building coroutines with the `yield` and `send()` statements. Then, we will look at different roles that coroutines can play in a pipeline, and how coroutines can support

multitasking.

5.3.1 Python Coroutines

In the previous section, we introduced generator functions, which use `yield` to return values. Python generator functions can also consume values using a `(yield)` statement. In addition two new methods on generator objects, `send()` and `close()`, create a framework for objects that *consume* and produce values. Generator functions that define these objects are coroutines.

Coroutines consume values using a `(yield)` statement as follows:

```
value = (yield)
```

With this syntax, execution pauses at this statement until the object's `send` method is invoked with an argument:

```
coroutine.send(data)
```

Then, execution resumes, with `value` being assigned to the value of `data`. To signal the end of a computation, we shut down a coroutine using the `close()` method. This raises a `GeneratorExit` exception inside the coroutine, which we can catch with a `try/except` clause.

The example below illustrates these concepts. It is a coroutine that prints strings that match a provided pattern.

```
>>> def match(pattern):
        print('Looking for ' + pattern)
        try:
            while True:
                s = (yield)
                if pattern in s:
                    print(s)
        except GeneratorExit:
            print("=== Done ===")
```

We initialize it with a pattern, and call `__next__()` to start execution:

```
>>> m = match("Jabberwock")
>>> m.__next__()
Looking for Jabberwock
```

The call to `__next__()` causes the body of the function to be executed, so the line "Looking for jabberwock" gets printed out. Execution continues until the statement `line = (yield)` is encountered. Then, execution pauses, and waits for a value to be sent to `m`. We can send values to it using `send`.

```
>>> m.send("the Jabberwock with eyes of flame")
the Jabberwock with eyes of flame
>>> m.send("came whiffling through the tulgey wood")
>>> m.send("and burbled as it came")
>>> m.close()
=== Done ===
```

When we call `m.send` with a value, evaluation resumes inside the coroutine `m` at the statement `line = (yield)`, where the sent value is assigned to the variable `line`. Evaluation continues inside `m`, printing out the line if it matches, going through the loop until it encounters `line = (yield)` again. Then, evaluation pauses inside `m` and resumes where `m.send` was called.

We can chain functions that `send()` and functions that `yield` together achieve complex behaviors. For example, the function below splits a string named `text` into words and sends each word to another coroutine.

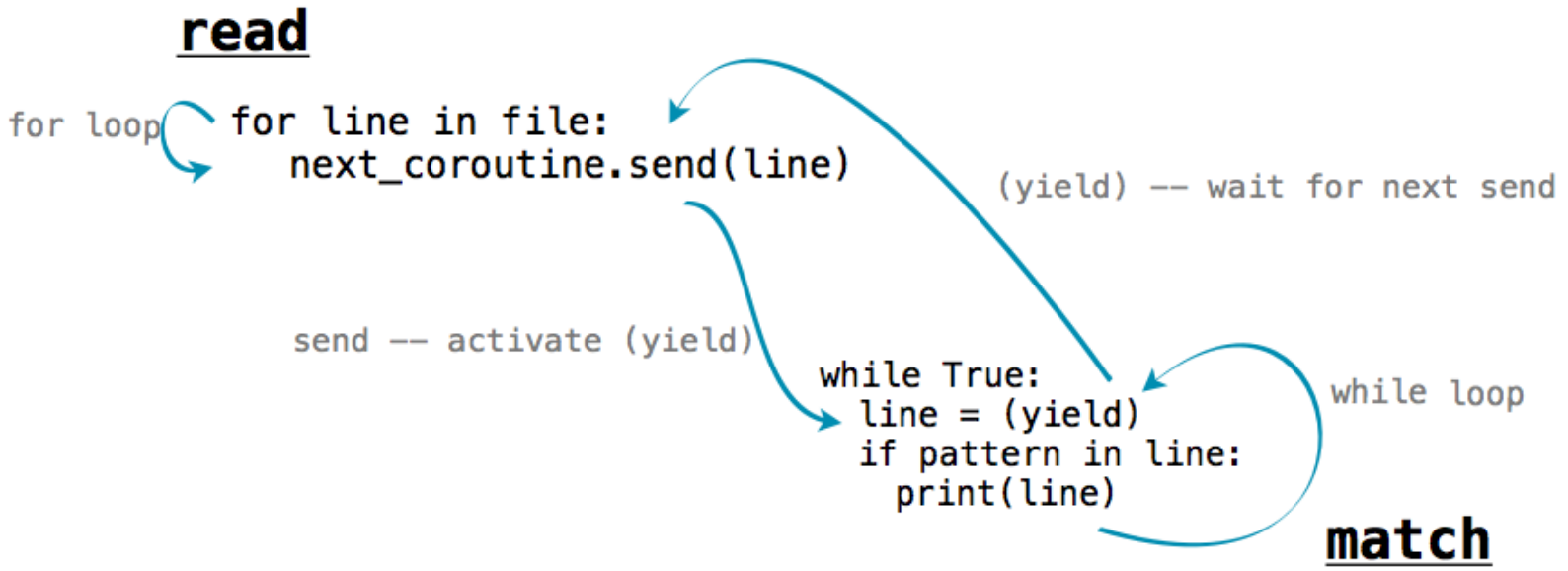
```
>>> def read(text, next_coroutine):
        for line in text.split():
            next_coroutine.send(line)
        next_coroutine.close()
```

Each word is sent to the coroutine bound to `next_coroutine`, causing `next_coroutine` to start executing, and this function to pause and wait. It waits until `next_coroutine` pauses, at which point the function resumes by sending the next word or completing.

If we chain this function together with `match` defined above, we can create a program that prints out only the words that match a particular word.

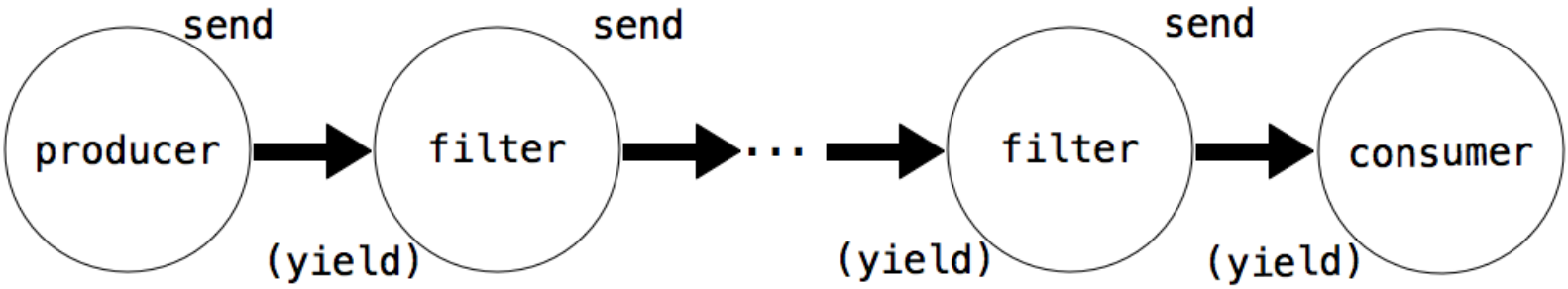
```
>>> text = 'Commending spending is offending to people pending lending!'
>>> matcher = match('ending')
>>> matcher.__next__()
Looking for ending
>>> read(text, matcher)
Commending
spending
offending
pending
lending!
=== Done ===
```

The `read` function sends each word to the coroutine `matcher`, which prints out any input that matches its `pattern`. Within the `matcher` coroutine, the line `s = (yield)` waits for each sent word, and it transfers control back to `read` when it is reached.



5.3.2 Produce, Filter, and Consume

Coroutines can have different roles depending on how they use `yield` and `send()`:



- A **Producer** creates items in a series and uses send(), but not `(yield)`
- A **Filter** uses `(yield)` to consume items and `send()` to send result to a next step.
- A **Consumer** uses `(yield)` to consume items, but does not send.

The function `read` above is an example of a *producer*. It does not use `(yield)`, but uses `send` to produce data items. The function `match` is an example of a consumer. It does not `send` anything, but consumes data with `(yield)`.We can break up `match` into a filter and a consumer. The filter would be a coroutine that only sends on strings that match its pattern.

```
>>> def match_filter(pattern, next_coroutine):
        print('Looking for ' + pattern)
        try:
            while True:
                s = (yield)
                if pattern in s:
                    next_coroutine.send(s)
        except GeneratorExit:
            next_coroutine.close()
```

And the consumer would be a function that printed out lines sent to it.

```
>>> def print_consumer():
        print('Preparing to print')
        try:
            while True:
                line = (yield)
                print(line)
        except GeneratorExit:
            print("=== Done ===")
```

When a filter or consumer is constructed, its `__next__` method must be invoked to start its execution.

```
>>> printer = print_consumer()
>>> printer.__next__()
Preparing to print
>>> matcher = match_filter('pend', printer)
>>> matcher.__next__()
Looking for pend
>>> read(text, matcher)
spending
pending
=== Done ===
```

Even though the name *filter* implies removing items, filters can transform items as well. The function below is an example of a filter that transforms items. It consumes strings and sends along a dictionary of the number of times different letters occur in the string.

```
>>> def count_letters(next_coroutine):
        try:
            while True:
                s = (yield)
                counts = {letter:s.count(letter) for letter in set(s)}
                next_coroutine.send(counts)
        except GeneratorExit as e:
            next_coroutine.close()
```

We can use it to count the most frequently-used letters in text using a consumer that adds up dictionaries and finds the most frequent key.

```
>>> def sum_dictionaries():
        total = {}
        try:
            while True:
                counts = (yield)
                for letter, count in counts.items():
                    total[letter] = count + total.get(letter, 0)
        except GeneratorExit:
            max_letter = max(total.items(), key=lambda t: t[1])[0]
            print("Most frequent letter: " + max_letter)
```

To run this pipeline on a file, we must first read the lines of a file one-by-one. Then, we send the results through `count_letters` and finally to `sum_dictionaries`. We can re-use the `read` coroutine to read the lines of a file.

```
>>> s = sum_dictionaries()
>>> s.__next__()
>>> c = count_letters(s)
>>> c.__next__()
>>> read(text, c)
Most frequent letter: n
```

5.3.3 Multitasking

A producer or filter does not have to be restricted to just one next step. It can have multiple coroutines downstream of it, and `send()` data to all of them. For example, here is a version of `read` that sends the words in a string to multiple next steps.

```
>>> def read_to_many(text, coroutines):
        for word in text.split():
            for coroutine in coroutines:
                coroutine.send(word)
        for coroutine in coroutines:
            coroutine.close()
```

We can use it to examine the same text for multiple words:

```
>>> m = match("mend")
>>> m.__next__()
Looking for mend
>>> p = match("pe")
>>> p.__next__()
Looking for pe
```

```
>>> read_to_many(text, [m, p])
Commending
spending
people
pending
=== Done ===
=== Done ===
```

First, `read_to_many` calls `send(word)` on `m`. The coroutine, which is waiting at `text = (yield)` runs through its loop, prints out a match if found, and resumes waiting for the next `send`. Execution then returns to `read_to_many`, which proceeds to send the same line to `p`. Thus, the words of `text` are printed in order.