4.2 Implicit Sequences

A sequence can be represented without each element being stored explicitly in the memory of the computer. That is, we can construct an object that provides access to all of the elements of some sequential dataset without computing the value of each element in advance. Instead, we compute elements on demand.

An example of this idea arises in the `range` container 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.

```
>>> 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. In computer science, *lazy computation* describes any program that delays the computation of a value until that value is needed.

4.2.1 Iterators

Python and many other programming languages provide a unified way to process elements of a container value sequentially, called an iterator. An *iterator* is an object that provides sequential access to values, one by one.

The iterator abstraction has two components: a mechanism for retrieving the next element in the sequence being processed and a mechanism for signaling that the end of the sequence has been reached and no further elements remain. For any container, such as a list or range, an iterator can be obtained by calling the built-in `iter` function. The contents of the iterator can be accessed by calling the built-in `next` function.

```
>>> primes = [2, 3, 5, 7]
>>> type(primes)
>>> iterator = iter(primes)
>>> type(iterator)
>>> next(iterator)
2
>>> next(iterator)
3
>>> next(iterator)
5
```

The way that Python signals that there are no more values available is to raise a `StopIteration` exception when `next` is called. This exception can be handled using a `try` statement.

```
>>> next(iterator)
7
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> try:
        next(iterator)
    except StopIteration:
        print('No more values')
No more values
```

An iterator maintains local state to represent its position in a sequence. Each time `next` is called, that position advances. Two separate iterators can track two different positions in the same sequence. However, two names for the same iterator will share a position, because they share the same value.

```
>>> r = range(3, 13)
>>> s = iter(r)  # 1st iterator over r
>>> next(s)
3
>>> next(s)
4
>>> t = iter(r)  # 2nd iterator over r
>>> next(t)
3
>>> next(t)
4
>>> u = t        # Alternate name for the 2nd iterator
>>> next(u)
5
>>> next(u)
6
```

Advancing the second iterator does not affect the first. Since the last value returned from the first iterator was 4, it is positioned to return 5 next. On the other hand, the second iterator is positioned to return 7 next.

```
>>> next(s)
5
>>> next(t)
7
```

Calling `iter` on an iterator will return that iterator, not a copy. This behavior is included in Python so that a programmer can call `iter` on a value to get an iterator without having to worry about whether it is an iterator or a container.

```
>>> v = iter(t)  # Another alterante name for the 2nd iterator
>>> next(v)
8
>>> next(u)
9
>>> next(t)
10
```

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, iterators are only required to 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 is often sufficient for data processing applications.

4.2.2 Iterables

Any value that can produce iterators is called an *iterable* value. In Python, an iterable value is anything that can be passed to the built-in `iter` function. Iterables include sequence values such as strings and tuples, as well as other containers such as sets and dictionaries. Iterators are also iterables, because they can be passed to the `iter` function.

Even unordered collections such as dictionaries must define an ordering over their contents when they produce iterators. Dictionaries and sets are unordered because the programmer has no control over the order of iteration, but Python does guarantee certain properties about their order in its specification.

TODO block quote

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d
{'one': 1, 'three': 3, 'two': 2}
>>> k = iter(d)
```

```
>>> next(k)
'one'
>>> next(k)
'three'
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
3
```

If a dictionary changes in structure because a key is added or removed, then all iterators become invalid and future iterators may exhibit arbitrary changes to the order their contents. On the other hand, changing the value of an existing key does not change the order of the contents or invalidate iterators.

```
>>> d.pop('two')
2
>>> next(k)

RuntimeError: dictionary changed size during iteration
Traceback (most recent call last):
```

4.2.3 Built-in Iterators

Several built-in functions take as arguments iterable values and return iterators. These functions are used extensively for lazy sequence processing.

The `map` function is lazy: calling it does not perform the computation required to compute elements of its result. Instead, an iterator object is created that can return results if queried using `next`. We can observe this fact in the following example, in which the call to `print` is delayed until the corresponding element is requested from the `doubled` iterator.

```
>>> def double_and_print(x):
        print('***', x, '=>', 2*x, '***')
        return 2*x
>>> s = range(3, 7)
>>> doubled = map(double_and_print, s)  # double_and_print not yet called
>>> next(doubled)                       # double_and_print called once
*** 3 => 6 ***
6
>>> next(doubled)                       # double_and_print called again
*** 4 => 8 ***
8
>>> list(doubled)                       # double_and_print called twice more
*** 5 => 10 ***
*** 6 => 12 ***
[10, 12]
```

The `filter` function returns an iterator over, `zip`, and `reversed` functions also return iterators.

TODO demonstrate these values

4.2.4 For Statements

The `for` statement in Python operates on iterators. Objects are *iterable* (an interface) if they have an `__iter__` method that returns an *iterator*. Iterable objects can be the value of the `<expression>` in the header of a `for` statement:

```
for <name> in <expression>:
    <suite>
```

To execute a `for` statement, Python evaluates the header `<expression>`, which must yield an iterable value. Then, the `__iter__` method is invoked on that value. Until a `StopIteration` exception is raised, Python repeatedly invokes the `__next__` method on that iterator and binds the result to the `<name>` in the `for` statement. Then, it executes the `<suite>`.

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

```
2
3
```

In the above example, the `counts` list returns an iterator from 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 execution of the `for` statement concludes.

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

```
>>> items = counts.__iter__()
>>> try:
        while True:
            item = items.__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 `items` 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.

To use an iterator in a for loop, the iterator must also have an `__iter__` method. The *Iterator types <http://docs.python.org/3/library/stdtypes.html#iterator-types> `_ section of the Python docs suggest that an iterator have an ``__iter__`` method that returns the iterator itself, so that all iterators are iterable.*

4.2.5 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__` method 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, the `yield` statement indicates that 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 invoked. The generator raises a `StopIteration` exception whenever its generator function returns.

4.2.6 Iterable Interface

An object is iterable if it returns an iterator when its `__iter__` method is invoked. Iterable values represent data collections, and they provide a fixed representation that may produce more than one iterator.

For example, an instance of the `Letters` class below represents a sequence of consecutive letters. Each time its `__iter__` method is invoked, a new `LetterIter` instance is constructed, which allows for sequential access to the contents of the sequence.

```
>>> class Letters:
        def __init__(self, start='a', end='e'):
            self.start = start
            self.end = end
        def __iter__(self):
            return LetterIter(self.start, self.end)
```

The built-in `iter` function invokes the `__iter__` method on its argument. In the sequence of expressions below, two iterators derived from the same iterable sequence independently yield letters in sequence.

```
>>> b_to_k = Letters('b', 'k')
>>> first_iterator = b_to_k.__iter__()
>>> next(first_iterator)
'b'
>>> next(first_iterator)
'c'
>>> second_iterator = iter(b_to_k)
>>> second_iterator.__next__()
'b'
>>> first_iterator.__next__()
'd'
>>> first_iterator.__next__()
'e'
>>> second_iterator.__next__()
'c'
>>> second_iterator.__next__()
'd'
```

The iterable `Letters` instance `b_to_k` and the `LetterIter` iterator instances `first_iterator` and `second_iterator` are different in that the `Letters` instance does not change, while the iterator instances do change with each call to `next` (or equivalently, each invocation of `__next__`). The iterator tracks progress through sequential data, while an iterable represents the data itself.

Many built-in functions in Python take iterable arguments and return iterators. The `map` function, for example, takes a function and an iterable. It returns an iterator over the result of applying the function argument to each element in the iterable argument.

```
>>> caps = map(lambda x: x.upper(), b_to_k)
>>> next(caps)
'B'
>>> next(caps)
'C'
```

4.2.7 Creating Iterables with Yield

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 invoked. 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* `LettersWithYield` class below returns a new iterator over letters each time `__iter__` is invoked.

```
>>> class LettersWithYield:
        def __init__(self, start='a', end='e'):
            self.start = start
            self.end = end
        def __iter__(self):
            next_letter = self.start
            while next_letter < self.end:
                yield next_letter
                next_letter = chr(ord(next_letter)+1)
```

The `__iter__` method is a generator function; it returns a generator object that yields the letters `'a'` through `'d'` and then stops. Each time we invoke this method, a new generator starts a fresh pass through the sequential data.

```
>>> letters = LettersWithYield()
>>> list(all_pairs(letters))[:5]
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'a')]
```

4.2.8 Iterator Interface

The Python iterator interface is defined using a method called `__next__` that returns the next element of some underlying sequential series that it represents. In response to invoking `__next__`, an iterator can perform arbitrary computation in order to either retrieve or compute the next element. 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 `LetterIter` class below iterates over an underlying series of letters from some `start` letter up to but not including some `end` letter. The instance attribute `next_letter` stores the next letter to be returned. The `__next__` method returns this letter and uses it to compute a new `next_letter`.

```
>>> class LetterIter:
        """An iterator over letters of the alphabet in ASCII order."""
        def __init__(self, start='a', end='e'):
```

```
            self.next_letter = start
            self.end = end
        def __next__(self):
            if self.next_letter == self.end:
                raise StopIteration
            letter = self.next_letter
            self.next_letter = chr(ord(letter)+1)
            return letter
```

Using this class, we can access letters in sequence using either the `__next__` method or the built-in `next` function, which invokes `__next__` on its argument.

```
>>> letter_iter = LetterIter()
>>> letter_iter.__next__()
'a'
>>> letter_iter.__next__()
'b'
>>> next(letter_iter)
'c'
>>> letter_iter.__next__()
'd'
>>> letter_iter.__next__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in next
StopIteration
```

Iterators are mutable: they track the position in some underlying sequence of values as they progress. When the end is reached, the iterator is used up. A `LetterIter` instance can only be iterated through once. After its `__next__()` method raises a `StopIteration` exception, it continues to do so from then on. Typically, an iterator is not reset; instead a new instance is created to start a new iteration.

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. The built-in `next` function in Python invokes the `__next__` method on its argument.

```
>>> class Positives:
        def __init__(self):
            self.next_positive = 1;
        def __next__(self):
            result = self.next_positive
            self.next_positive += 1
            return result
>>> p = Positives()
>>> next(p)
1
>>> next(p)
2
>>> next(p)
3
```

4.2.9 Streams

TODO

4.2.10 Python Streams

Streams offer another way to represent sequential data implicitly. A stream is a lazily computed linked list. Like the `Link` class from Chapter 2, a `Stream` instance responds to requests for its `first` element and the `rest` of the stream. Like an `Link`, the `rest` of a `Stream` is itself a `Stream`. Unlike an `Link`, 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 the rest explicitly.

```
>>> class Stream:
        """A lazily computed linked list."""
        class empty:
            def __repr__(self):
                return 'Stream.empty'
        empty = empty()
        def __init__(self, first, compute_rest=lambda: empty):
            assert callable(compute_rest), 'compute_rest must be callable.'
            self.first = first
            self._compute_rest = compute_rest
        @property
        def rest(self):
            """Return the rest of the stream, computing it if necessary."""
            if self._compute_rest is not None:
                self._rest = self._compute_rest()
                self._compute_rest = None
            return self._rest
        def __repr__(self):
            return 'Stream({0}, <...>)'.format(repr(self.first))
```

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

```
>>> r = Link(1, Link(2+3, Link(9)))
```

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. We achieve this effect by creating anonymous functions.

```
>>> s = Stream(1, lambda: Stream(2+3, lambda: Stream(9)))
```

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.

Accessing the elements of linked 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
Link(5, Link(9))
>>> s.rest
Stream(5, <...>)
```

While the `rest` of `r` is a two-element linked 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._rest` is `None`, signifying that the rest of the `Stream` has not yet been computed. When the `rest` attribute is requested via a dot expression, the `rest` property 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, then discarded.

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

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 integer_stream(first):
        def compute_rest():
            return integer_stream(first+1)
        return Stream(first, compute_rest)

>>> positives = integer_stream(1)
>>> positives
Stream(1, <...>)
```

```
>>> positives.first
1
```

When `integer_stream` is called for the first time, it returns a stream whose `first` is the first integer in the sequence. However, `integer_stream` is actually recursive because this stream's `compute_rest` calls `integer_stream` again, with an incremented argument. We say that `integer_stream` is lazy because the recursive call to `integer_stream` is only made whenever the `rest` of an integer stream is requested.

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

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 is Stream.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 is Stream.empty:
            return s
        def compute_rest():
            return filter_stream(fn, s.rest)
        if fn(s.first):
            return Stream(s.first, compute_rest)
        else:
            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 coerce up to the first `k` elements to a Python `list`.

```
>>> def first_k_as_list(s, k):
        first_k = []
        while s is not Stream.empty and k > 0:
            first_k.append(s.first)
            s, k = s.rest, k-1
        return first_k
```

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

```
>>> s = integer_stream(3)
>>> s
Stream(3, <...>)
>>> m = map_stream(lambda x: x*x, s)
>>> m
Stream(9, <...>)
>>> first_k_as_list(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.

```
>>> prime_numbers = primes(integer_stream(2))
>>> first_k_as_list(prime_numbers, 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 `prime_numbers` is still 2 after converting the prefix of the stream to a list.

```
>>> prime_numbers.first
2
```

Just as linked 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.

Continue: 4.3 Declarative Programming