ITERATORS, GENERATORS, AND STREAMS 10

COMPUTER SCIENCE 61A

April 16, 2015

1 Iterators

An *iterator* is an object that represents a sequence of values. Here is an example of a class that implements Python's iterator interface. This iterator calculates all of the natural numbers one-by-one, starting from zero:

```
class Naturals():
    def __init__(self):
        self.current = 0
    def __next__(self):
        result = self.current
        self.current += 1
        return result
    def __iter__(self):
        return self
```

There are two components of Python's iterator interface: the `__next__` method, and the `__iter__` method.

1.1 `__next__`

The `__next__` method usually does two things:

1. calculates the next value
2. checks if it has any values left to compute

To return the next value in the sequence, the iterator does some computation defined in the `__next__` method.

When there are no more values left to compute, the `__next__` method must raise a type of exception called `StopIteration`. This signals the end of the sequence.

Note: the `__next__` method defined above does NOT raise any `StopIteration` exceptions. Why? Because there are always more values left to compute! Remember, there is no "last natural number", so there is technically no "end of the sequence." However, if you wanted to define a *finite* iterator, then you would raise a `StopIteration` after returning the final value.

1.2 `__iter__`

The purpose of the `__iter__` method is to return an iterator object. By definition, an iterator object is an object that has implemented both the `__next__` and `__iter__` methods.

This has an interesting consequence. If a class implements both a `__next__` method and a `__iter__` method, its `__iter__` method can just return `self` (like in the example). Since the class implements both `__next__` and `__iter__`, it is technically an iterator object, so its `__iter__` method can just return itself.

1.3 Implementation

When defining an iterator object, you should always keep track of how much of the sequence has already been computed. In the above example, we use an instance variable `self.current` to keep track.

Iterator objects maintain state. Successive calls to `__next__` will most likely output different values each time, so `__next__` is considered *non-pure*.

How do we call `__next__` and `__iter__`? Python has built-in functions called `next` and `iter` for this. Calling `next(some_iterator)` will then cause Python to implicitly call `some_iterator`'s `__next__` method. Calling `iter(some_iterator)` will make a similar implicit call to `some_iterator`'s `__iter__` method.

For example, this is how we would use the `Naturals` iterator:

```
>>> nats = Naturals()
>>> nats_iter = iter(nats)
>>> next(nats_iter)
0
>>> next(nats_iter)
1
>>> next(nats_iter)
2
```

However, we don't really need to call `iter` on `nats`. Why not?

Because you can use iterator objects in `for` loops. In other words, any object that satisfies the iterator interface can be iterated over:

```
>>> nats = Naturals()
>>> for n in nats:
        print(n)
0
1
2
...  # Forever!
```

This works because the Python `for` loop implicitly calls the `__iter__` method of the object being iterated over, and repeatedly calls `next` on it. In other words, the above interaction is (basically) equivalent to:

```
nats_iter = iter(nats)
is_done = False
while not is_done:
    try:
        val = next(nats_iter)
        print(val)
    except StopIteration:
        is_done = True
```

1.4 Questions

1. Define an iterator whose i-th element is the result of combining the i-th elements of two input iterables using some binary operator, also given as input. The resulting iterator should have a size equal to the size of the shorter of its two input iterators.

   ```
   >>> from operator import add
   >>> evens = IterCombiner(Naturals(), Naturals(), add)
   >>> next(evens)
   0
   >>> next(evens)
   2
   >>> next(evens)
   4
   ```

```
class IterCombiner(object):
    def __init__(self, iter1, iter2, combiner):
```

Solution:

```
        self.iter1 = iter(iter1)
        self.iter2 = iter(iter2)
        self.combiner = combiner
```

```
    def __next__(self):
```

Solution:

```
        return self.combiner(next(self.iter1), next(self.
            iter2))
```

```
    def __iter__(self):
```

Solution:

```
        return self
```

2. What is the result of executing this sequence of commands?

```
>>> naturals = Naturals()
>>> doubled_naturals = IterCombiner(naturals, naturals, add)
>>> next(doubled_naturals)
```

Solution: 1

```
>>> next(doubled_naturals)
```

Solution: 5

1.5 Extra Practice

1. Create an iterator that generates the sequence of Fibonacci numbers.

```
class Fibonacci(object):
    def __init__(self):
```

Solution:

```
        self.current = 0
        self.next = 1
```

```
    def __next__(self):
```

Solution:

```
        res = self.current
        self.current, self.next = self.next, self.current +
            self.next
        return res
```

```
    def __iter__(self):
```

Solution:

```
        return self
```

2 Generators

A **generator** function is a special kind of Python function that uses a `yield` statement instead of a `return` statement to report values. When a generator function is called, it returns an iterable object.

Here is an iterator for the natural numbers written using the generator construct:

```
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
```

Calling `generate_naturals()` will return a generator object:

```
>>> gen = generate_naturals()
>>> gen
<generator object gen at ...>
```

To use the generator object, you then call `next` on it:

```
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
```

Think of a generator object as containing an implicit `__next__` method. This means, by definition, a generator object is an iterator.

2.1 `yield`

The `yield` statement is similar to a `return` statement. However, while a `return` statement closes the current frame after the function exits, a yield statement causes the frame to be saved until the next time `__next__` is called, which allows the generator to automatically keep track of the iteration state.

Once `__next__` is called again, execution picks up from where the previously executed `yield` statement left off, and continues until the next `yield` statement (or the end of the function) is encountered.

Including a `yield` statement in a function automatically signals to Python that this function will create a generator. When we call the function, it will return a generator object, instead of executing the code inside the body. When the returned generator's `__next__` method is called, the code in the body is executed for the first time, and stops executing upon reaching the first `yield` statement.

2.2 Implementation

Because generators are technically iterators, you can implement `__iter__` methods using only generators. For example,

```
class Naturals():
    def __init__(self):
        self.current = 0
    def __iter__(self):
        while True:
            yield self.current
```

```
            self.current += 1
```

Naturals __iter__ method now returns a generator object. The usage of a `Naturals` object is exactly the same as before:

```
>>> nats = Naturals()
>>> nats_iter = iter(nats)
>>> next(nats_iter)
0
>>> next(nats_iter)
1
>>> next(nats_iter)
2
```

There are a couple of things to note:

- *No `__next__` method in `Naturals`*. Remember, __iter__ only needs to return an object that has implemented a __next__ method. Since generators have their own __next__ method, the new `Naturals` implementation is perfectly valid.
- *`nats` is a `Naturals` object and `nats_iter` is a generator*

Since generators are iterators, you can also use generators in `for` loops.

2.3 Questions

1. Define a generator that yields the sequence of perfect squares.

```
def perfect_squares():
```

Solution:

```
i = 0
while True:
    yield i * i
    i += 1
```

2.4 Extra Practice

1. Write a generator function that returns lists of all subsets of the positive integers from 1 to n. Each call to this generator's __next__ method will return a list of subsets of the set `[1, 2, ..., n]`, where `n` is the number of times __next__ was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> next(subsets)
    [[]]
    >>> next(subsets)
    [[], [1]]
    >>> next(subsets)
    [[], [1], [2], [1, 2]]
    """
```

Solution:

```
subsets = [[]]
n = 1
while True:
    yield subsets
    subsets = subsets + [s + [n] for s in subsets]
    n += 1
```

3 Streams

A *stream* is a lazily-evaluated linked list. A stream's elements (except for the first element) are only computed when those values are needed.

```
class Stream:
    class empty:
        """An empty stream"""
    empty = empty()

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        assert callable(compute_rest), 'must be a function'
        self.first = first
        self._compute_rest = compute_rest

    @property
    def rest(self):
        """Return the rest, computing it if necessary."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest
```

A `Stream` instance is similar to a `Link` instance. Both have `first` and `rest` attributes. The rest of a `Link` is either a `Link` or `Link.empty`. Likewise, the rest of a `Stream` is either a `Stream` or `Stream.empty`.

However, instead of specifying all of the elements in `__init__`, we provide a function, `compute_rest`, that will be called to compute the remaining elements of the stream. Remember that the code in the function body is not evaluated until it is called, which lets us implement the desired evaluation behavior.

This implementation of streams also uses *memoization*. The first time a program asks a `Stream` for its `rest` field, the `Stream` code computes the required value using `compute_rest`, saves the resulting value, and then returns it. After that, every time the `rest` field is referenced, the stored value is simply returned.

Here is an example:

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

Here, we start out with a stream whose first element is 1, and whose `compute_rest` function creates another stream. So when we do compute the `rest`, we get another stream whose first element is one greater than the previous element, and whose `compute_rest` creates another stream. Hence, we effectively get an infinite stream of integers, computed one at a time. This is almost like an infinite recursion, but one which can be viewed one step at a time, and so does not crash.

3.1 Questions

1. Suppose one wants to define a random infinite stream of numbers via the recursive definition: "a random infinite stream consists of a first random number, followed by a remaining random infinite stream." Consider an attempt to implement this via the code. Are there any problems with this? How can we fix this?

```
from random import random
random_stream = Stream(random(), lambda: random_stream)
```

Solution: The provided code will generate a single random number, and then produce an infinite stream which simply repeats that one number over and over. To fix this, we can make this into a function that returns a Stream:

```
def random_stream():
    return Stream(random(), random_stream)
```

2. Write a function `every_other`, which takes in an infinite stream and returns a stream containing its even indexed elements.

```
def every_other(s):
```

Solution:

```
    return Stream(s.first, lambda: every_other(s.rest.rest)
       )
```

3.2 Extra Questions

1. Write a function `fib_stream` that creates an infinite stream of Fibonacci Numbers, using the `add_streams` function that was introduced in lab.

```
def fib_stream():
```

Solution:

```
def fib_stream():
    def compute_rest():
        return add_streams(fib_stream(),
                           fib_stream().rest)
    return Stream(0, lambda: Stream(1, compute_rest))
```

2. Write a function `seventh` that creates an infinite stream of the decimal expansion of dividing n by 7.

```
def seventh(n):
    """The decimal expansion of n divided by 7.

    >>> first_k(seventh(1), 10)
    [1, 4, 2, 8, 5, 7, 1, 4, 2, 8]
    """
```

Solution:

```
    q, r = n*10 // 7, n*10 % 7
    return Stream(q, lambda: seventh(r))
```

3.3 Higher-Order Functions on Streams

Stream processing functions can be higher-order, abstracting a general computational process over streams. Take a look at `filter_stream`:

```
def filter_stream(filter_func, s):
    def compute_rest():
        return filter_stream(filter_func, s.rest)

    if s is Stream.empty:
        return s
    elif filter_func(s.first):
        return Stream(s.first, compute_rest)
    else:
        return compute_rest()
```

The Stream we create has as its `compute_rest` a function that "promises" to filter the rest of the Stream when called. So at any one point, the entire stream has not been filtered. Instead, only the part that has been referenced has been filtered.

3.4 Questions

1. What does the following Stream output? Try writing out the first few values of the stream to see the pattern.

```
def my_stream():
    def compute_rest():
        return add_streams(map_stream(double, my_stream()),
                           my_stream())
    return Stream(1, compute_rest)
```

> **Solution:** Powers of 3: 1, 3, 9, 27, 81, ...

2. (Summer 2012 Final) What are the first five values in the following stream?

```
def my_stream():
    def compute_rest():
        return add_streams(filter_stream(lambda x: x % 2 == 0,
               my_stream()), map_stream(lambda x: x + 2,
                  my_stream()))
    return Stream(2, compute_rest)
```

Solution: 2, 6, 14, 30, 62