CS61A Lab 13: Lab 13: Streams, Iterators, Generators

Week 13, 2012

This week, we'll be exploring methods of creating sequences of values incrementally, using three different constructs. We first start with iterators, which we've already seen earlier in the course, and then we will introduce generators and streams.

Iterators

Recall that, in Python, an iterator is a kind of object that is generally used in for loops like so:

```
for item in sequence:
    # use item
```

Which is equivalent to the following loop:

```
iterator = iter(sequence)
while True:
    try:
        item = next(iterator)
    except StopIteration:
        break # leave the loop
    # use item
```

An iterator is the object returned from the `__iter__` method of a sequence you'd like to iterate over. An iterator implements the `__next__` method which, each time it is called, either will return the next item in the sequence or will raise a `StopIteration` exception to indicate that it has no more values to return.

Generators

In Python, generators are often used as an alternative way to create a sequence of values. A generator is created by calling a function that uses the `yield` statement at one or more points to produce a value, instead of using the ordinary `return` statement. Each time the generator is invoked after having produced a value with `yield`, it will pick up from the line where it last left off, rather than starting over from the beginning. An example is the following function, which defines a generator producing the sequence of positive integers:

```
def integers():
    x = 1
    while True:
        yield x
        x += 1
```

The presence of the line `yield x` tells Python that this function describes a generator. Furthermore, whenever the interpreter reaches that line, it will produce `x` as a value and stop running the generator; the generator will later resume from that point in the code if/when called again to ask for another value. To actually use a generator, we can call the function defining it in a for loop in the same position where we might otherwise have used a sequence:

```
>>> for i in integers():
...     print(i)
...
1
2
3
4
etc.
```

If we want a generator to finish producing values, we simply have the function exit. As an example we have the generator `integers_to` below:

```
def integers_to(n):
    current = 1
    while current <= n:
        yield current
        current += 1
    # The function stops here
```

Question #0:

Write a generator `FibYield` which produces the elements of the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, ..., one by one, never stopping.

Streams

Streams are another way to represent a sequence of values and are often useful for representing "infinite" sequences. One basic implementation of a Stream is the following class definition:

```
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):
        assert not self.empty, ’Empty streams have no rest.’
        if not self._computed:
            self._rest = self._compute_rest()
            self._computed = True
        return self._rest

empty_stream = Stream(None, None, True)
```

A Stream looks much like an RList in that it has a first item and a "rest", a reference to a Stream of the remaining items. However, unlike an RList, a Stream is "lazy"[1]: it does not immediately compute all of the values in the sequence, instead generally waiting until we ask for an item before actually computing its value. (Furthermore, you might notice that Streams, as implemented here, involve something like memoization: once we've computed a value we remember it and reuse it later whenever asked for it again.)

The power of Streams is that we don't immediately figure out all of the values until we need them; as a result we can "store" an infinite sequence of items because we don't explicitly enumerate all the items in the computer. (Note that the definition used above allows instances of Stream to represent finite sequences, as well. In the exact same way as for infinite sequences, we can also "store" a finite, but very long, sequence of items without using a proportional amount of memory, because we needn't actually explicitly store all the items ahead of time; all that need be stored is a description of how to compute them as needed)

In lecture, we saw how to define for Streams the analogues of various familiar list processing functions; for example, `map_stream`, `filter_stream`, and `combine_streams`. These can make working with Streams much more convenient and readable. Let us implement some more functions of this sort:

Question #1:

Define a function `Find` which takes in as input a stream and a predicate, and returns the first element in the stream satisfying the predicate. [What if there is no element in the stream satisfying the predicate? Could we add to the specification that this function must throw an exception in that case? Explain the difficulty implementing such a specification, and the conditions under which one can and cannot meet it.]

Question #2:

Define a function `Cycle` which takes in as input an ordinary Python list and returns a stream which just repeats that list over and over (E.g., `Cycle(['a', 'b', 'c'])` is the stream `<'a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c', ...>`). If the input list is empty, output the empty stream; otherwise, the output stream should be infinite.

Question #3:

Define a function `PartialSums`, which takes in a stream <a_1, a_2,a_3, ...> and outputs the stream <a_1, $a_1 + a_2$, $a_1 + a_2 + a_3$, ...>. [If the input is a finite stream of length n, the output should be a finite stream of length n. If the input is an infinite stream, the output should also be an infinite stream].

Consider `p = PartialSums(Cycle([1]))`. What do you think the stream `p` looks like? How can you use the Python interpreter to check? [You may want to keep in mind some of the conveniences already defined in lecture]

Question #4:

Define a function `Interleave`, which takes in two streams <a_1, a_2, a_3, ...> and <b_1, b_2, b_3, ...> and returns the new stream <a_1, b_1, a_2, b_2, a_3, b_3, ...>. If either of the inputs is finite, the output stream should be finite as well, terminating just at the point where it would be impossible to go on. If both of the inputs are infinite, the output stream should be infinite as well.

Question #5:

What do you think the sequence produced by `combine_streams(lambda x, y: 4 * x - y, Interleave(Cycle([0], p), p)` looks like? Assign this stream to `oddly_tripled_p` and use the Python interpreter to check. Can you find a different way to produce this same sequence? (There are many! Try finding a few, using the various stream operations we've developed in lecture and lab so far)

[Just for fun[2]: Define `staggered_double(x)` to be `combine_streams(add, x, Stream(0, lambda:x))` and define `twelve_p` to be `staggered_double(staggered_double(oddly_tripled_p))`. What does `twelve_p` look like? The answer may surprise you with its cleanliness! This observation is used to fruitfully assign a (somewhat counterintuitive) value to the sum of `p` in suitable contexts in mathematics]

Question #6:

Recall that our Stream class remembers elements it has previously computed and does not recompute them. This can play in unexpected ways with computations whose values may change over time (so-called "impure" computations).

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:

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

What it is unsatisfactory about this? How can one fix it?

Lab is over. Here come the footnotes...

[1]: Hence my selection to write this lab. Laziness is my wheelhouse. -SR↩

[2]: My fun, not yours. -SR↩