Iterators

An iterator is an object that tracks the position in a sequence of values in order to provide sequential access. It returns elements one at a time and is only good for one pass through the sequence. The following is an example of a class that implements Python’s iterator interface using two special methods `__next__` and `__iter__`. This iterator calculates all of the natural numbers one-by-one, starting from zero:

```python
class Naturals():
    def __init__(self):
        self.current = 0

    def __next__(self):
        result = self.current
        self.current += 1
        return result

    def __iter__(self):
        return self
```

1.1 `__next__`

The `__next__` method checks if it has any values left in the sequence; if it does, it computes the next element. To return the next value in the sequence, the `__next__` method keeps track of its current position in the sequence. If there are no more values left to
compute, it must raise an exception called `StopIteration`. This signals the end of the sequence.

*Note:* the `__next__` method defined in the `Naturals` class does *not* raise `StopIteration` because there is no “last natural number”.

### 1.2 `__iter__`

The `__iter__` method returns an iterator object. If a class implements both a `__next__` method and an `__iter__` method, its `__iter__` method can simply return `self` as the class itself is an iterator. In fact, Python specifies that an iterator’s `__iter__` method should return `self`.

### 1.3 Iterables

An *iterable* object represents a sequence. Examples of iterables are lists, tuples, strings, and dictionaries. The iterable class must implement an `__iter__` method, which returns an iterator. Note that since all iterators have an `__iter__` method, they are all iterable.

In general, a sequence’s `__iter__` method will return a new iterator every time it is called. This is because an iterator cannot be reset. Returning a new iterator allows us to iterate through the same sequence multiple times.

### 1.4 Implementation

When defining an iterator, you should always keep track of current position in the sequence. In the `Naturals` class, we use `self.current` to save the position.

Iterator objects maintain state. Each successive call to `__next__` will return the next element in the sequence. Since this element may be different from the previous one, `__next__` is considered *non-pure*.

Python has built-in functions called `next` and `iter` that call `__next__` and `__iter__` respectively.

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

```python
>>> nats = Naturals()
>>> next(nats)
0
>>> next(nats)
1
>>> next(nats)
2
```
1.5 Questions

1. Define an iterator whose $i$th element is the result of combining the $i$th elements of two input iterators 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.

```python
>>> from operator import add
>>> evens = IteratorCombiner(Naturals(), Naturals(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4
class IteratorCombiner(object):
    def __init__(self, iterator1, iterator2, combiner):
        self.iterator1 = iterator1
        self.iterator2 = iterator2
        self.combiner = combiner

    def __next__(self):
        try:
            value1 = next(self.iterator1)
            value2 = next(self.iterator2)
            return self.combiner(value1, value2)
        except StopIteration:
            raise StopIteration

    def __iter__(self):
        return self
```

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

```python
>>> nats = Naturals()
>>> doubled_nats = IteratorCombiner(nats, nats, add)
>>> next(doubled_nats)
0
>>> next(doubled_nats)
2
>>> next(doubled_nats)
4
```
1.6 Extra Question

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

```python
class FibIterator(object):
    def __init__(self):
        pass

    def __next__(self):
        pass

    def __iter__(self):
        return self
```

2. Streams

2.1 Introduction

In Python, we can use iterators and generators to represent infinite sequences. However, Scheme does not support iterators. Let’s see what happens when we use a Scheme list to represent an infinite sequence of natural numbers.

```scheme
(scm> (define (naturals n)
        (cons n (naturals (+ n 1))))

naturals

scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because the second argument to `cons` is always evaluated, we cannot create an infinite sequence of integers using a Scheme list.

Instead, we have extended our Scheme interpreter (and scheme.cs61a.org) to support `streams`, which are `lazy` Scheme lists. The first element is represented explicitly, but the rest of the stream’s elements are computed only when needed. This evaluation strategy, where we don’t compute a value until it is needed, is called `lazy evaluation`. Let’s try to implement the sequence of natural numbers again using a stream!

```scheme
(scm> (define (naturals n)
        (cons-stream n (naturals (+ n 1))))

naturals
```
We use the special form `cons-stream` to create a stream. Note that `cons-stream` is not a procedure, because the second operand `(naturals (+ n 1))` is not evaluated when `cons-stream` is called. It’s only evaluated when `cdr-stream` is used to inspect the rest of the stream.

Here are some primitives pertaining to streams:

- `nil` is the empty stream
- `cons-stream` creates a non-empty stream from an initial element and an expression to compute the rest of the stream
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

Streams are very similar to Scheme lists. The `cdr` of a Scheme list is either another Scheme list or `nil`; likewise, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that the expression for the rest of the stream is computed the first time that `cdr-stream` is called, instead of when `cons-stream` is used. Subsequent calls to `cdr-stream` return this value without recomputing it. This allows us to efficiently work with infinite streams like the `naturals` example above. We can see this in action by using a non-pure function to compute the rest of the stream:

```
scm> (define (compute-rest n)
...>   (print "evaluating!"
...>   (cons-stream n nil))
compute-rest
s
s
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
"evaluating!"
1
scm> (car (cdr-stream s))
1
```

Note that the string "evaluating!" is only printed the first time `cdr-stream` is called, and no other time.
2.2 Questions

1. What would Scheme print?
   The following function has been defined for you:

   ```scheme
   scm> (define (has-even? s)
          (cond ((null? s) False)
                 ((even? (car s)) True)
                 (else (has-even? (cdr-stream s))))
   has-even?
   scm> (define ones (cons-stream 1 ones))
   scm> (define twos (cons-stream 2 twos))
   scm> ones
   scm> (cdr ones)
   scm> (cdr-stream ones)
   scm> (has-even? ones)
   scm> (has-even? twos)
   ```

2. Write `map-stream`, which takes a function \( f \) and a stream \( s \) and returns a new stream, which has all the elements from \( s \), but with \( f \) applied to each one.

   ```scheme
   (define (map-stream f s)
   ```

   ```scheme
   scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
   evens
   scm> (cdr-stream evens)
   (2 . #[promise (not forced)])
   ```
3. Using streams can be tricky! Compare the following two implementations of `filter-stream`, the first is a correct implementation whereas the second is wrong in some way. What’s wrong with the second implementation?

; Correct
(define (filter-stream f s)
  (if (null? s)
    nil
    (if (f (car s))
      (cons-stream (car s)
                   (filter-stream f (cdr-stream s)))
      (filter-stream f (cdr-stream s))))

; Incorrect
(define (filter-stream f s)
  (if (null? s)
    nil
    (let ((rest (filter-stream f (cdr-stream s))))
      (if (f (car s))
        (cons-stream (car s) rest)
        rest))))

4. Write a function `slice` which takes in a `stream`, a `start`, and an `end`. It should return a Scheme list that contains the elements of `stream` between index `start` and `end`, not including `end`. If the stream ends before `end`, you can return `nil`.

```
(define (slice stream start end)
  ...
)
```

sclm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
5. The Fibonacci sequence is a classic infinite sequence. Implement `make-fib-stream`, which takes two numbers and produces a stream of Fibonacci numbers starting with those two numbers.

```scheme
(define (make-fib-stream a b)
  scm> (define fib-stream (make-fib-stream 0 1))
  fib-stream
  scm> (slice fib-stream 0 10)
  (0 1 1 2 3 5 8 13 21 34)
)

6. Since streams only evaluate the next element when they are needed, we can combine infinite streams together for interesting results! We’ve defined the function `zip-with` for you below. Use it to define a few of our favorite sequences.

```scheme
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys))
      nil
      (cons-stream
       (f (car xs) (car ys))
       (zip-with f (cdr-stream xs) (cdr-stream ys)))))

scm> (define evens (zip-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
(define factorials

scm> (slice factorials 0 10)
(1 1 2 6 24 120 5040 40320 362880)
(define fibs

scm> (slice fibs 0 10)
(0 1 1 2 3 5 8 13 21 34)
```
2.3 Extra Questions

1. Write a function `range-stream` which takes a start and end argument, and returns a stream that represents the integers between included start and end - 1.

   `(define (range-stream start end) ...)

2. We can even represent the sequence of all prime numbers as an infinite stream! Define a function `sieve`, which takes in a stream of increasing numbers and returns a stream containing only those numbers which are not multiples of an earlier number in the stream. We can define `primes` by sifting all natural numbers starting at 2. Look online for the Sieve of Eratosthenes if you need some inspiration.

   `(define (sieve s) ...

   `(define primes
      (sieve (naturals 2)))

   scm> (slice primes 0 10)
   (2 3 5 7 11 13 17 19 23 29)`