ITERATORS, GENERATORS, AND STREAMS

COMPUTER SCIENCE MENTORS 61A

November 7 to November 11, 2016

1 Iterators

1. What is difference between an iterable and an iterator?

   Solution: Iterator: Mutable object that tracks a position in a sequence, advancing on each call to `next`

   Iterable: Represents a sequence and returns a new iterator on each call to `iter`

   To use in an English sentence: Lists are "iterable". To go through a list, you make an object called an "iterator" to scan through the list.

2. **Accumulator** Write an iterator class that takes in a list and calculates the sum of the list thus far.

```
>>> accu = Accumulator([1, 2, 3, 4, 5, 6])
>>> for a in accu:
...     print(a)
1
3
6
10
15
21
```

   Solution:

```
class Accumulator:
    def __init__(self, lst):
```

```
        self.lst = lst
        self.index = 0
        self.sum = 0
    def __next__(self):
        if self.index >= len(self.lst):
            raise StopIteration()
        self.sum += self.lst[self.index]
        self.index += 1
        return self.sum
    def __iter__(self):
        return self
```

3. Is this an iterator or an iterable or both?

Solution: Both; the `iter` method returns self and the `next` method is implemented. Note that an iterator is always an iterable, but an iterable is not always an iterator.

4. (Optional) Write `Accumulator` so it works if it takes in any iterable, not just a list

Solution:

```
class Accumulator:
    def __init__(self, iterable):
            self.iterable = iterable
        self.iterator = iter(iterable)
            self.sum = 0
    def __next__(self):
        self.sum += next(self.iterator)
            return self.sum
    def __iter__(self):
            return self
```

2 Generators

1. What does the following code block output?

```
def foo():
    a = 0
    if a < 10:
```

```
        print("Hello")
        yield a
        print("World")

for i in foo():
    print(i)
```

Solution:

```
Hello
0
World
```

First time you call foo, it will yield a (which starts as 0)

2. How can we modify `foo` so that `list(foo()) == [1, 2, 3, . . . , 10]`? (It's ok if there are extra `print`s)

Solution: Change the `if` to a `while` statement, and make sure to increment a. This looks like:

```
def foo():
    a = 0
    while a < 10:
        a += 1
        yield a
```

3. Define `hailstone_sequence` a generator that yields the hailstone sequence. Remember, for the hailstone sequence, if n is even, we need to divide by two, otherwise, we will multiply by 3 and add by 1.

```
; Doctests:
>>> hs_gen = hailstone_sequence(10)
>>> hs_gen.__next__()
10
>>> next(hs_gen) #equivalent to previous
5
>>> for i in hs_gen:
>>> print(i)
16
8
4
2
1
```

Solution:

```
def hailstone_sequence(n):
    while n != 1:
        yield n
        if n % 2 == 0:
            n = n // 2
        else:
            n = n*3 + 1
    yield n
```

4. (Optional) Define `tree_sequence`, a generator that iterates through a tree by first yielding the root value and then yield each branch.

```
>>> tree = Tree(1, [Tree(2, [Tree(5)]), Tree(3, [Tree(4)])])
>>> print(list(tree_sequence(tree)))
[1, 2, 5, 3, 4]
```

Solution:

```
def tree_sequence(tree):
        yield tree.entry
        for branch in tree.branches:
                for value in tree_sequence(branch):
                        yield value
```

3 Streams

1. Whats the advantage of using a stream over a linked list?

   Solution: Lazy evaluation. We only evaluate up to what we need.

2. Whats the maximum size of a stream?

   Solution: Infinity

3. Whats stored in first and rest? What are their types?

   Solution: First is a value, rest is another stream (either a method to calculate it, or an already calculated stream). In the case of Scheme, this is called a promise.

4. When is the next element actually calculated?

   Solution: Only when it's requested (and hasn't already been calculated)

5. For each of the following lines of code, write what Scheme would output.

```
scm> (define x 1)
```

   Solution: `x`

```
scm> (if 2 3 4)
```

   Solution: `3`

```
scm> (delay (+ x 1))
```

   Solution:
   `#[promise]`

```
scm> (define (foo x) (+ x 10))
```

   Solution: `foo`

```
scm> (define bar (cons-stream (foo 1) (cons-stream (foo 2)
   bar)))
```

Solution: `bar`

```
scm> (car bar)
```

Solution: `11`

```
scm> (cdr bar)
```

Solution:
`#[promise]`

```
scm> (define (foo x) (+ x 1))
```

Solution: `foo`

```
scm> (cdr-stream bar)
```

Solution:
`(3 . #[promise])`

```
scm> (define (foo x) (+ x 5))
```

Solution: `foo`

```
scm> (car bar)
```

Solution: `11`

```
scm> (cdr-stream bar)
```

Solution:
`(3 . #[promise])`

6. Write out `double_naturals`, which is a stream that evaluates to the sequence 1, 1, 2, 2, 3, 3, etc.

```
(define (double_naturals)
    (double_naturals_helper 1 0)
)

(define (double_naturals_helper first flag)
```

```
)
```

Solution:

```
(define (double_naturals_helper first flag)
    (if (= 1 flag)
        (cons-stream first (double_naturals_helper (+ 1
           first) 0))
        (cons-stream first (double_naturals_helper first
           1))
    )
)

;Alternative Solutions
(define (double_naturals_helper first flag)
    (cons-stream first (double_naturals_helper (+ flag
       first) (- 1 flag)))
)
```

7. Write out `interleave`, which returns a stream that alternates between the values in stream1 and stream2. Assume that the streams are infinitely long.

```
(define (interleave stream1 stream2)
```

)

Solution:

```
(define (interleave stream1 stream2)
    (cons-stream
        (car stream1)
        (interleave stream2 (cdr-stream stream1))
    )
)

(define (interleave stream1 stream2)
    (cons-stream (car stream1)
        (cons-stream (car stream2)
            (interleave (cdr-stream stream1) (cdr-stream
              stream2))))
)
```