CS 61A Iterators, Generators, and Streams
Spring 2019 Discussion 10: April 17, 2019 Solutions

1 Iterators and Generators

An **iterable** is a data type which contains a collection of values which can be processed one by one sequentially. Some examples of iterables we've seen include lists, tuples, strings, and dictionaries. In general, any object that can be iterated over in a `for` loop can be considered an iterable.

```
>>> a = [1, 2]
>>> a_iter = iter(a)
>>> next(a_iter)
1
>>> next(a_iter)
2
>>> next(a_iter)
StopIteration
```

While an iterable contains values that can be iterated over, we need another type of object called an **iterator** to actually retrieve values contained in an iterable. Calling the `iter` function on an iterable will create an iterator over that iterable. Each iterator keeps track of its position within the iterable. Calling the `next` function on an iterator will give the current value in the iterable and move the iterator's position to the next value.

In this way, the relationship between an iterable and an iterator is analogous to the relationship between a book and a bookmark - an iterable contains the data that is being iterated over, and an iterator keeps track of your position within that data.

Once an iterator has returned all the values in an iterable, subsequent calls to `next` on that iterable will result in a `StopIteration` exception. In order to be able to access the values in the iterable a second time, you would have to create a second iterator.

One important application of iterables and iterators is the `for` loop. We've seen how we can use `for` loops to iterate over iterables like lists and dictionaries.

```
counts = [1, 2, 3]

for i in counts:
    print(i)

items = iter(counts)
while True:
    try:
        i = next(items)
        print(i)
    except StopIteration:
        break #Exit the while loop
```

This only works because the `for` loop implicitly creates an iterator using the built-in `iter` function. Python then calls `next` repeatedly on the iterator, until it raises `StopIteration`.

The code to the right shows how we can mimic the behavior of `for` loops using `while` loops.

Note that most iterators are also iterables - that is, calling `iter` on them will return an iterator. This means that we can use them inside `for` loops. However, calling `iter` on most iterators will not create a new iterator - instead, it will simply return the same iterator.

We can also iterate over iterables in a list comprehension or pass in an iterable to the built-in function `list` in order to put the items of an iterable into a list.

In addition to the sequences we've learned, Python has some built-in ways to create iterables and iterators. Here are a few useful ones:

- `range(start, end)` returns an iterable containing numbers from start to end-1. If `start` is not provided, it defaults to 0.

- `map(f, iterable)` returns a new iterator containing the values resulting from applying `f` to each value in `iterable`.
- `filter(f, iterable)` returns a new iterator containing only the values in `iterable` for which `f(value)` returns `True`.

Questions

1.1 What would Python display? If a StopIteration Exception occurs, write `StopIteration`, and if another error occurs, write `Error`.

```
>>> lst = [6, 1, "a"]
>>> next(lst)
```

```
Error
```

```
>>> lst_iter = iter(lst)
>>> next(lst_iter)
```

```
6
```

```
>>> next(lst_iter)
```

```
1
```

```
>>> next(iter(lst))
```

```
6
```

```
>>> [x for x in lst_iter]
```

```
["a"]
```

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 a generator object, which is a type of iterator.* To the right, you can see a function that returns an iterator over the natural numbers. 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 resumes where it last stopped and continues until the next **`yield`** statement or the end of the function. A generator function can have multiple **`yield`** statements.

```
>>> def gen_naturals():
...     current = 0
...     while True:
...         yield current
...         current += 1
>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

Including a **`yield`** statement in a function automatically tells Python that this function will create a generator. When we call the function, it returns a generator object instead of executing the body. When the generator's **`next`** method is called, the body is executed until the next **`yield`** statement is executed.

When **yield from** is called on an iterator, it will **yield** every value from that iterator. It's similar to doing the following:

```
for x in an_iterator:
    yield x
```

The example to the right demonstrates different ways of computing the same result.

```
>>> square = lambda x: x*x
>>> def many_squares(s):
...     for x in s:
...         yield square(x)
...     yield from map(square, s)
...
>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9]
```

Questions

1.1 What would Python display? If a StopIteration Exception occurs, write `StopIteration`, or if another error occurs, write `Error`.

```
>>> def weird_gen(x):
...     if x % 2 == 0:
...        yield x * 2
...     else:
...        yield x
...        yield from weird_gen(x - 1)
>>> next(weird_gen(2))
```

```
4
```

```
>>> list(weird_gen(3))
```

```
[3, 4]
```

```
>>> def greeter(x):
...     while x % 2 != 0:
...         print('hello!')
...         yield x
...         print('goodbye!')
>>> greeter(5)
```

```
<generator object greeter at ...>
```

```
>>> gen = greeter(5)
>>> next(gen)
```

```
hello!
5
```

```
>>> next(gen)
```

```
goodbye!
hello!
5
```

1.2 Implement `filter_link`, which takes in a linked list `link` and a function `f` and returns a generator which yields the values of `link` for which `f` returns `True`.

Try to implement this both using a while loop and without using any form of iteration.

```
def filter_link(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> g = filter_link(link, lambda x: x % 2 == 0)
    >>> next(g)
    2
    >>> next(g)
    StopIteration
    >>> list(filter_link(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    while _________________________:

        if ________________________:

            _______________________

        ___________________________
```

```
def filter_link(link, f):
    while link is not Link.empty:
        if f(link.first):
            yield link.first
        link = link.rest
```

```
def filter_no_iter(link, f):
    """
    >>> link = Link(1, Link(2, Link(3)))
    >>> list(filter_no_iter(link, lambda x: x % 2 != 0))
    [1, 3]
    """

    if ____________________________:

        return

    elif __________________________:

        ___________________________

    _______________________________
```

```
def filter_no_iter(link, f):
    if link is Link.empty:
        return
    elif f(link.first):
        yield link.first
    yield from filter_no_iter(link.rest, f)
```

1.3 Implement `sum_paths_gen`, which takes in a Tree instance `t` and and returns a generator which yields the sum of all the nodes from a path from the root of a tree to a leaf.

You may yield the sums in any order.

```
def sum_paths_gen(t):
    """
    >>> t1 = Tree(5)
    >>> next(sum_paths_gen(t1))
    5
    >>> t2 = Tree(1, [Tree(2, [Tree(3), Tree(4)]), Tree(9)])
    >>> sorted(sum_paths_gen(t2))
    [6, 7, 10]
    """

    if ___________________________:

        yield ____________________

    for __________________________:

        for __________________________:

            yield ____________________
```

```
def sum_paths_gen(t):
    if t.is_leaf():
        yield t.label
    for b in t.branches:
        for s in sum_paths_gen(b):
            yield s + t.label
```

2 Streams

In Python, we can use iterators to represent infinite sequences (for example, the generator for all natural numbers). However, Scheme does not support iterators. Let's see what happens when we try to use a Scheme list to represent an infinite sequence of natural numbers:

```
scm> (define (naturals n)
         (cons n (naturals (+ n 1))))
naturals
scm> (naturals 0)
Error: maximum recursion depth exceeded
```

Because `cons` is a regular procedure and both its operands must be evaluted before the pair is constructed, we cannot create an infinite sequence of integers using a Scheme list.

Instead, our Scheme interpreter supports *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. Computing a value only when it's needed is also known as *lazy evaluation*.

```
scm> (define (naturals n)
         (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
scm> (cdr nat)
#[promise (not forced)]
scm> (car (cdr-stream nat))
1
scm> (car (cdr-stream (cdr-stream nat)))
2
```

We use the special form `cons-stream` to create a stream:

```
(cons-stream <operand1> <operand2>)
```

`cons-stream` is a special form because the second operand is not evaluated when evaluating the expression. To evaluate this expression, Scheme does the following:

1. Evaluate the first operand.
2. Construct a promise containing the second operand.
3. Return a pair containing the value of the first operand and the promise.

To actually get the rest of the stream, we must call `cdr-stream` on it to force the promise to be evaluated. Note that this argument is only evaluated once and is then stored in the promise; subsequent calls to `cdr-stream` returns the 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
scm> (define s (cons-stream 0 (compute-rest 1)))
s
scm> (car (cdr-stream s))
evaluating!
1
scm> (car (cdr-stream s))
1
```

Here, the expression `compute-rest 1` is only evaluated the first time `cons-stream` is called, so the symbol `evaluating!` is only printed the first time.

When displaying a stream, the first element of the stream and the promise are displayed separated by a dot (this indicates that they are part of the same pair, with the promise as the `cdr`). If the value in the promise has not been evaluated by calling `cdr-stream`, we consider it to be not forced. Otherwise, we consider it forced.

```
scm> (define s (cons-stream 1 nil))
s
scm> s
(1 . #[promise (not forced)])
scm> (cdr-stream s) ; nil
()
scm> s
(1 . #[promise (forced)])
```

Streams are very similar to Scheme lists in that they are also recursive structures. Just like the `cdr` of a Scheme list is either another Scheme list or `nil`, the `cdr-stream` of a stream is either a stream or `nil`. The difference is that whereas both arguments to `cons` are evaluated upon calling `cons`, the second argument to `cons-stream` isn't evaluated until the first time that `cdr-stream` is called.

Here's a summary of what we just went over:

- `nil` is the empty stream
- `cons-stream` constructs a stream containing the value of the first operand and a promise to evaluate the second operand
- `car` returns the first element of the stream
- `cdr-stream` computes and returns the rest of stream

Video walkthrough

Questions

2.1 What would Scheme display?

As you work through these problems, remember that streams have two important components:

- Lazy evaluation – so the remainder of the stream isn't computed until explicitly requested.
- Memoization – so anything we compute won't be recomputed.

The examples here stretch these concepts to the limit. In most practical use cases, you may find you rarely need to redefine functions that compute the remainder of the stream.

```
scm> (define (has-even? s)
       (cond ((null? s) #f)
             ((even? (car s)) #t)
             (else (has-even? (cdr-stream s)))))
has-even?
scm> (define (f x) (* 3 x))
f
scm> (define nums (cons-stream 1 (cons-stream (f 3) (cons-stream (f 5) nil))))
nums

scm> nums

(1 . #[promise (not forced)])

scm> (cdr-stream nums)

(9 . #[promise (not forced)])

scm> nums

(1 . #[promise (forced)])

scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)

(9 . #[promise (not forced)])

scm> (cdr-stream (cdr-stream nums))

(10 . #[promise (not forced)])

scm> (has-even? nums)

True
```

Video walkthrough

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

```
(define (slice s start end)
```

```
  (cond
    ((or (null? s) (= end 0)) nil)
    ((> start 0)
      (slice (cdr-stream s) (- start 1) (- end 1)))
    (else
      (cons (car s)
        (slice (cdr-stream s) (- start 1) (- end 1))))))
```

```
scm> (define nat (naturals 0)) ; See naturals procedure defined earlier
nat
scm> (slice nat 4 12)
(4 5 6 7 8 9 10 11)
```

2.3 Since streams only evaluate the next element when they are needed, we can combine infinite streams together for interesting results! Use it to define a few of our favorite sequences. We've defined the function `combine-with` for you below, as well as an example of how to use it to define the stream of even numbers.

```
(define (combine-with f xs ys)
  (if (or (null? xs) (null? ys))
    nil
    (cons-stream
      (f (car xs) (car ys))
      (combine-with f (cdr-stream xs) (cdr-stream ys)))))
scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens
scm> (slice evens 0 10)
(0 2 4 6 8 10 12 14 16 18)
```

For these questions, you may use the `naturals` stream in addition to `combine-with`.

i.
```
(define factorials

  (cons-stream 1 (combine-with * (naturals 1) factorials)))

scm> (slice factorials 0 10)
(1 1 2 6 24 120 720 5040 40320 362880)
```

(Continued on next page)

ii.
```
(define fibs

  (cons-stream 0
    (cons-stream 1
      (combine-with + fibs (cdr-stream fibs)))))

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

iii. (Extra for practice) Write `exp`, which returns a stream where the nth term represents the degree-n polynomial expantion for e^x, which is $\sum_{i=0}^{n} x^i/i!$.

You may use `factorials` in addition to `combine-with` and `naturals` in your solution.

```
(define (exp x)

  (let ((terms (combine-with (lambda (a b) (/ (expt x a) b))
                             (cdr-stream (naturals 0))
                             (cdr-stream factorials))))
    (cons-stream 1 (combine-with + terms (exp x)))))

scm> (slice (exp 2) 0 5)
(1 3 5 6.333333333 7 7.266666667)
```