1 Iterables and Iterators

An iterable object is any container that can be processed sequentially. Examples of iterables are lists, tuples, strings, and dictionaries. To process the elements sequentially, call iter on the iterable to retrieve an iterator.

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. To access the next element of an iterator, call next on the object. Each time next is called, the iterator advances.

We can create as many iterators as we would like from a single iterable. However, iterators will go through the elements of the sequence they represent only once. To go through an iterable twice, create two iterators!

```python
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)
4
```
1.1 For Loops

We have already been using iterables to go through the elements of a sequence. This happens all the time in for loops. For example:
```python
>>> for n in [1, 2, 3]:
  ...  print(n)
... 1
2
3
```
This works because the for loop implicitly creates an iterator using the `iter` method. Python then repeatedly calls `next` repeatedly on the iterator, until it raises `StopIteration`. In other words, the loop above is (basically) equivalent to:
```python
iterator = iter([1, 2, 3])
try:
    while True:
        n = next(iterator)
        print(n)
except StopIteration:
    pass
```

1.2 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 iterator. The following is a function that returns an iterator for the natural numbers:
```python
def gen_naturals():
    current = 0
    while True:
        yield current
        current += 1
```
Calling `generate_naturals()` will return a generator object, which you can use to retrieve values.
```python
>>> gen = gen_naturals()
>>> gen
<generator object gen at ...
```
1.3 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 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.

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.

1.4 yield from

The `yield from` statement is similar to a `yield` statement. `yield from` takes in an iterator and yields each of the values from that iterator. It can be used in conjunction with other `yield` and `yield from` statements.

```python
>>> square = lambda x: x*x
>>> def many_squares(s):
...     for x in s:
...         yield square(x)
...         yield from [square(x) for x in s]
...         yield from map(square, s)
...

>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9, 1, 4, 9]
```

When the `list` function in Python receives an iterator, it calls the `next` function on the input until it raises a `StopIteration`. It puts each of the elements from the calls to `next` into a new list and returns it.

1.5 Questions

1. Define an generator function that combines two input iterators using a given combiner function. 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 = combiner(gen_naturals(), gen_naturals(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4
def combiner(iterator1, iterator2, combiner):

2. What is the result of executing this sequence of commands?
>>> nats = gen_naturals()
>>> doubled_nats = combiner(nats, nats, add)
>>> next(doubled_nats)

>>> next(doubled_nats)

3. Write a generator function that returns 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, \ldots, n\}, where n is the number of times next was previously called.
def generate_subsets():
    
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]

    """
In Python, we can use iterators to represent infinite sequences. 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 the second argument to `cons` is always evaluated, we cannot create an infinite sequence of integers using a Scheme list.

Instead, our Scheme interpreter (and scheme.cs61a.org) 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. 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!

```
scm> (define (naturals n)
       (cons-stream n (naturals (+ n 1))))
naturals
scm> (define nat (naturals 0))
nat
scm> (car nat)
0
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. Note that `cons-stream` is a special form, 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.

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

```scheme
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
```

Note that the symbol \texttt{evaluating!} is only printed the first time \texttt{cdr-stream} is called.

### 2.1 Questions

1. What would Scheme display?

```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 \texttt{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. 
\begin{verbatim}
(define (map-stream f s)

scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
\end{verbatim}

3. Using streams can be tricky! Compare the following two implementations of \texttt{filter-stream}, the first is a correct implementation whereas the second is wrong in some way. What’s wrong with the second implementation?
\begin{verbatim}
; Correct
(define (filter-stream f s)
  (cond
   ((null? s) nil)
   ((f (car s)) (cons-stream (car s) (filter-stream f
                                     (cdr-stream s))))
   (else (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))))
\end{verbatim}

4. Write a function \texttt{range-stream} which takes a \texttt{start} and \texttt{end} argument, and returns a stream that represents the integers between included \texttt{start} and \texttt{end} - 1.
\begin{verbatim}
(define (range-stream start end)

scm> (define odds (range-stream 1 5))
odds
scm> (car (cdr-stream odds))
1
\end{verbatim}
5. 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.

```scheme
(define (slice stream start end)
  scm> (slice nat 4 12)
  (4 5 6 7 8 9 10 11)

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)