1 Iterables and Iterators

An iterable is any container that can be processed sequentially. Examples include lists, tuples, strings, and dictionaries. Often we want to access the elements of an iterable, one at a time. We find ourselves writing \texttt{lst[0]}, \texttt{lst[1]}, \texttt{lst[2]}, and so on. It would be more convenient if there was an object that could do this for us, so that we don’t have to keep track of the indices.

This is where iterators come in. Given an iterable, we can call the \texttt{iter} function on that iterable to return a new iterator object. Each time we call \texttt{next} on the iterator object, it gives us one element at a time, just like we wanted. When it runs out of elements to give, calling \texttt{next} on the iterator object will raise a \texttt{StopIteration} exception.

We can create as many iterators as we would like from a single iterable. But, each iterator goes through the elements of the iterable only once. If you want to go through an iterable twice, create two iterators!

For Loops

By now, you are familiar with using \texttt{for} loops to iterate over iterables like lists and dictionaries.

This only works because the \texttt{for} loop implicitly creates an iterator using the built-in \texttt{iter} function. Python then calls \texttt{next} repeatedly on the iterator, until it raises \texttt{StopIteration}. The code to the right is (basically) equivalent to using a \texttt{for} loop to iterate over a list of \texttt{[1, 2, 3]}.

Other Iterable Uses

We have already encountered functions that use and create iterables. Here are some that we have seen (and some that we have not):

- \texttt{range(start, end)} - Creates iterable of numbers from start (inclusive) to end (exclusive)
- \texttt{map(f, iterable)} - Creates iterator over \( f(x) \) for \( x \) in \texttt{iterable}
- \texttt{filter(f, iterable)} - Creates iterator over \( x \) for \( x \) in \texttt{iterable} if \( f(x) \)
- \texttt{zip(iter1, iter2)} - Creates iterator over co-indexed pairs \( (x, y) \)
- \texttt{reversed(iterable)} - Creates iterator sequence in reverse order
- \texttt{list(iterable)} - Creates a list containing all \( x \) in \texttt{iterable}
- \texttt{tuple(iterable)} - Creates a tuple containing all \( x \) in \texttt{iterable}
• `sorted(iterable)` - Creates a sorted list containing all x in iterable

Questions

1.1 What would Python display?

It can be helpful to refer back to the `iter` example on the first page. Remember that calling `iter` returns something that you can call `next` on. The rest of the challenge in this problem is just keeping track of where you currently are in the sequence.

```python
>>> lst = [[1, 2]]
>>> i = iter(lst)
>>> j = iter(next(i))
>>> next(j)
1
>>> lst.append(3)
>>> next(i)
3
>>> next(j)
2
>>> next(i)
StopIteration
```

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.* To the right, you can see a function that returns an iterator over the natural numbers.

2.1 `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.

```python
>>> def gen_naturals():
...     current = 0
...     while True:
...         yield current
...         current += 1
>>> gen = gen_naturals()
>>> next(gen)
0
>>> next(gen)
1
```

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.* To the right, you can see a function that returns an iterator over the natural numbers.

```python
>>> 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, 1, 4, 9]
```
When `yield from` is called on an iterator, it will `yield` every value from that iterator. It’s similar to doing the following:

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

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

2.1 Write a generator function `combiner` 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
```

```python
def combiner(iterator1, iterator2, combiner):
    while True:
        yield combiner(next(iterator1), next(iterator2))
```

While this is the most compact solution, it may not be immediately obvious that we would arrive at this. It’s acceptable to start with the “basic skeleton” of all generators:

```python
while True:
    <do some work here>
    yield <something>
    <do some other work here>
```

From this, we put in some basic steps:
- We want to fetch the next item from both our iterators.
- Then, we would want to combine them using our combiner function.
- Finally, we want to yield the result (be very careful not to return!).

```python
while True:
    n1 = next(iterator1)
    n2 = next(iterator2)
    result = combiner(n1, n2)
```

2.2 What is the result of executing this sequence of commands?

```python
>>> nats = gen_naturals()
>>> doubled_nats = combiner(nats, nats, add)
>>> next(doubled_nats)
1
```

```python
>>> next(doubled_nats)
5
```
The naturals iterator has been fed into `combiner` twice. So the first yield will get the first two numbers out of naturals, the second yield will the the third and fourth numbers, and so on.

- $0 + 1 = 1$
- $2 + 3 = 5$

If you expected this to return 0 then 2, think about what would need to be changed in how we use `combiner`. Also, let’s assume we don’t want to change the behaviour of the `combiner` function.

2.3 Write a generator function `gen_all_items` that takes a list of iterators and yields items from all of them in order.

```python
def gen_all_items(lst):
    """
    >>> nums = [[1, 2], [3, 4], [[5, 6]]]
    >>> num_iters = [iter(l) for l in nums]
    >>> list(gen_all_items(num_iters))
    [1, 2, 3, 4, [5, 6]]
    """
    
    for it in lst:
        yield from it
```

The `yield from` solution is nice and short. But this can also be done with just `yield`:

```python
for it in lst:
    for x in it:
        yield x
```

Notice that this function will not yield out of deep lists. That is, it will keep the brackets around deep lists and yield them together instead of one element at a time.

For an extra challenge, figure out how to yield deep list items! (so the example in the doctest would return `[1, 2, 3, 4, 5, 6]`)
2.4 Write a generator function `generate_subsets` 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 previous calls to `next`.

```python
def generate_subsets():
    
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
        [[]]
        [[], [1]]
        [[], [1], [2], [1, 2]]
    
subsets = [[]
    n = 1
while True:
    yield subsets
    subsets = subsets + [s + [n] for s in subsets]
    n += 1
```

We start with a base list of subsets. To get the next sequence of subsets, we need two things:

- All current subsets will continue to be valid subsets in the future.
- We take all the subsets we currently have, and add the next number. These are also valid subsets.

3 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:

```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, 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*.
Delayed Expressions

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

```scheme
(define nat (naturals 0))
nat
car nat
0
car (cdr-stream nat)
1
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 `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:

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

Note that the symbol `evaluating!` is only printed the first time `cdr-stream` is called.

Questions

3.1 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.

```scheme
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 nums)
#\[promise (not forced)]
scm> (cdr-stream nums)
(9 . #\[promise (not forced)])
scm> (define (f x) (* 2 x))
f
scm> (cdr-stream nums)
(9 . #\[promise (not forced)])
scm> (has-even? nums)
True
```

3.2 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
```scheme
(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
```scheme
(define (filter-stream f s)
Delayed Expressions

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

Evaluating rest will result in infinite recursion if s is an infinite stream! In the
body of filter-stream, rest is always computed before cons-stream can delay the
evaluation.

Another way of thinking about this is that everything in the body of the let doesn’t
matter. All we will be doing is repeatedly doing the recursive call on filter-stream.

3.3 Write a function map-stream, which takes a function f and a stream s. It returns a
new stream which has all the elements from s, but with f applied to each one.

(define (map-stream f s)
  (if (null? s)
    nil
    (cons-stream (f (car s)) (map-stream f (cdr-stream s)))))

It might help to also compare this to the version of map for regular (non-stream)
Scheme lists:

(define (map f s)
  (if (null? s)
    nil
    (cons (f (car s)) (map f (cdr s)))))

Not too different, eh? The main change we’ve made is indicating we want to lazily
evaluate the rest of our stream by using cons-stream instead of cons, and recog-
nizing is a stream rather than a regular list by using cdr-stream.

scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
3.4 Write a function \texttt{range-stream} which takes a \texttt{start} and \texttt{end}, and returns a stream that represents the integers between \texttt{start} and \texttt{end - 1} (inclusive).

\begin{verbatim}
(define (range-stream start end)
  (if (= start end)
      nil
      (cons-stream start (range-stream (+ start 1) end))))
\end{verbatim}

It might help to compare this to the version of \texttt{range} for regular (non-stream) Scheme lists:

\begin{verbatim}
(define (range start end)
  (if (= start end)
      nil
      (cons start (range (+ start 1) end))))
\end{verbatim}

\texttt{scm> (define s (range-stream 1 5))}
\texttt{s}
\texttt{scm> (car (cdr-stream s))}
\texttt{2}

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

\begin{verbatim}
(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))))))
\end{verbatim}

\texttt{scm> (slice nat 4 12)}
\texttt{(4 5 6 7 8 9 10 11)}

3.6 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 \texttt{combine-with} for you below, as well as an example of how to use it to define the stream of even numbers.

\begin{verbatim}
(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)))))
\end{verbatim}

\texttt{scm> (define evens (combine-with + (naturals 0) (naturals 0)))
evens}
\texttt{scm> (slice evens 0 10)}
\texttt{(0 2 4 6 8 10 12 14 16 18)}

For these questions, you may use the \texttt{naturals} stream in addition to \texttt{combine-with}.
Delayed Expressions

i. \( \textbf{(define factorials} \) \)

\[
(\text{cons-stream 1 (combine-with } \ast \text{ (naturals 1) factorials}))
\]

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

ii. \( \textbf{(define fibs} \) \)

\[
(\text{cons-stream 0}
(\text{cons-stream 1}
(\text{combine-with + fibs (cdr-stream fibs)})))
\]

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

iii. Write \textbf{exp}, which returns a stream where the \( n \)th term represents the degree-\( n \) polynomial expansion for \( e^x \), which is \( \sum_{i=0}^{n} \frac{x^i}{i!} \).

You may use \textbf{factorials} in addition to \textbf{combine-with} and \textbf{naturals} in your solution.

\( \textbf{(define (exp x)} \) \)

\[
\text{let } ((\text{terms (combine-with (lambda (a b) (/ (expt x a) b))}
(\text{cdr-stream (naturals 0)}))
(\text{cdr-stream factorials})))))
(\text{cons-stream 1 (combine-with + terms (exp x))})
\]

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