CS 61A
Fall 2017

Delayed Expressions

Discussion 9: November 8, 2017

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 `lst[0]`, `lst[1]`, `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 **`iter`** function on that iterable to return a new iterator object. Each time we call **`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 **`next`** on the iterator object will raise a `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!

```
>>> lst = [4, 2]
>>> i = iter(lst)
>>> j = iter(lst)
>>> i
<list_iterator object>
>>> next(i)
4
>>> next(i)
2
>>> next(j)
4
>>> next(i)
StopIteration
>>> next(j)
2
```

For Loops

By now, you are familiar with using **`for`** loops to iterate over iterables like lists and dictionaries.

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 is (basically) equivalent to using a **`for`** loop to iterate over a list of `[1, 2, 3]`.

```
>>> counts = [1, 2, 3]
>>> items = iter(counts)
>>> try:
...     while True:
...         item = next(items)
...         print(item)
... except StopIteration:
...     pass # Do nothing
...
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):

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

- `sorted(iterable)` - Creates a sorted list containing all x in iterable

Questions

1.1 What would Python display?

```
>>> lst = [[1, 2]]
>>> i = iter(lst)
>>> j = iter(next(i))
>>> next(j)

>>> lst.append(3)
>>> next(i)

>>> next(j)

>>> next(i)
```

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.

```
>>> 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
```

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 the body. When the generator's **next** method is called, the body is executed until the next **yield** statement is executed.

2.2 yield from

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
```

```
>>> 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]
```

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.

```
>>> 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):
```

2.2 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)
```

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

```
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]]
    """
```

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, ..., n]`, where `n` is the number of previous calls to `next`.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```

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:

```
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*.

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

Questions

3.1 What would Scheme display?

```
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

scm> (cdr nums)

scm> (cdr-stream nums)

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

scm> (has-even? nums)
```

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
(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))))
```

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)
```

```
scm> (define evens (map-stream (lambda (x) (* x 2)) nat))
evens
scm> (car (cdr-stream evens))
2
```

3.4 Write a function `range-stream` which takes a `start` and `end`, and returns a stream that represents the integers between `start` and `end - 1` (inclusive).

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

```
scm> (define s (range-stream 1 5))
s
scm> (car (cdr-stream s))
2
```

3.5 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)
```

```
scm> (slice nat 4 12)
(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 `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`.

(Continued on the next page)

i. `(define factorials`

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

ii. `(define fibs`

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

iii. 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)`

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