A stream is our third example of a lazy sequence. A stream is like a lazily evaluated Rlist. In other words, the stream’s elements (except for the first element) are only evaluated when the values are needed.

Take a look at the following code:

```python
class Stream:
    class empty:
        pass
    empty = empty()

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        self.first = first
        self._compute_rest = compute_rest

    @property
    def rest(self):
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest
```

We represent Streams using Python objects, similar to the way we defined Rlists. We nest streams inside one another, and compute one element of the sequence at a time.

Note that instead of specifying all of the elements in `__init__`, we provide a function, `compute_rest`, that encapsulates the code to calculate the remaining elements of the
stream. Remember that the code in the function body is not evaluated until it is called, which lets us implement the desired evaluation behavior.

This implementation of streams also uses memoization. The first time a program asks a Stream for its rest field, the Stream code computes the required value using compute_rest, saves the resulting value, and then returns it. After that, every time the rest field is referenced, the stored value is simply returned.

Here is an example:

```python
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```

Notice what is happening here. We start out with a stream whose first element is 1, and whose compute_rest function creates another stream. So when we do compute the rest, we get another stream whose first element is one greater than the previous element, and whose compute_rest creates another stream. Hence, we effectively get an infinite stream of integers, computed one at a time. This is almost like an infinite recursion, but one which can be viewed one step at a time, and so does not crash.

### 1.1 Questions

1. Write a procedure `make_fib_stream()` that creates an infinite stream of Fibonacci Numbers. Make the first two elements of the stream 0 and 1. **Hint:** Consider using a helper function that can take two arguments, then think about how to start calling that function. Alternatively, you can implement it using the `add_streams()` function that was introduced in lecture.

```python
def make_fib_stream():
```

2. Suppose one wants to define a random infinite stream of numbers via the recursive definition: “a random infinite stream consists of a first random number, followed by a remaining random infinite stream.” Consider an attempt to implement this via the code. Are there any problems with this? How can we fix this?

```python
from random import random
random_stream = Stream(random(), lambda: random_stream)
```
1.2 Higher Order Functions on Streams

Naturally, as the theme has always been in this class, we can abstract our stream procedures to be higher order. Take a look at filter_stream:

```python
def filter_stream(filter_func, s):  
    def make_filtered_rest():  
        return filter_stream(filter_func, s.rest)  
    if s is Stream.empty:  
        return s  
    elif filter_func(s.first):  
        return Stream(s.first, make_filtered_rest)  
    else:  
        return filter_stream(filter_func, s.rest)
```

You can see how this function might be useful. Notice how the Stream we create has as its compute_rest function a procedure that “promises” to filter out the rest of the Stream when asked. So at any one point, the entire stream has not been filtered. Instead, only the part of the stream that has been referenced has been filtered, but the rest will be filtered when asked. We can model other higher order Stream procedures after this one, and we can combine our higher order Stream procedures to do incredible things!

1.3 Questions

1. What does the following Stream output? Try writing out the first few values of the stream to see the pattern.

```python
def my_stream():  
    def compute_rest():  
        return add_streams(map_stream(double, my_stream()), my_stream())  
    return Stream(1, compute_rest)
```

2. (Summer 2012 Final) What are the first five values in the following stream?

```python
def my_stream():  
    def compute_rest():  
        return add_streams(stream_filter(lambda x: x%2 == 0, my_stream()), stream_map(lambda x: x+2, my_stream()))  
    return Stream(2, compute_rest)
```
3. In a similar model to filter_stream, let’s recreate the procedure map_stream from lecture, that given a stream stream and a one-argument function func, returns a new stream that is the result of applying func on every element in s.

```python
def stream_map(func, s):
```

2 Review

It’s never too early to start reviewing for the final. Hurray!

1. (Fall 2011 Final) Implement a generator function, unique, that takes an iterable argument and returns an iterator over all the unique elements of its input in the order in which they first appear. Do not use any def, for, or class statements, or lambda expressions.

```python
def unique(iterable):
    """
    >>> list(unique([1, 3, 2, 2, 5, 3, 4, 1]))
    [1, 3, 2, 5, 4]
    """
```
2. (Fall 2012 Final) Implement a reversed relationship in logic. You may assume that an append-to-form relationship exists.

```logic
(fact (append-to-form () ?x ?x))
(fact (append-to-form (?a . ?r) ?y (?a . ?z))
    (append-to-form ?r ?y ?z))
```

3. (Spring 2013 Final) Write a relation sorted that is true if the given list is sorted in increasing order. Assume that you have a <= relation that relates two items if the first is less than or equal to the second.

```logic
(fact (<= a a))
(fact (<= a b))
(fact (<= b b))
(fact (sorted ()))
```

```
Success!
(query (sorted (a b)))
Success!
(query (sorted (b a)))
Failed.
```

4. (Spring 2012 Final) A classic puzzle called the Towers of Hanoi is a game that consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.
The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Complete the definition of `towers_of_hanoi` which prints out the steps to solve this puzzle for any number of \( n \) disks starting from the `start` rod and moving them to the `end` rod:

```python
def move_disk(start, end):
    print("Move 1 disk from rod", start, "to rod", end)

def towers_of_hanoi(n, start, end):
    """Print the moves required to solve the towers of hanoi game if we start with \( n \) disks on the start pole and want to move them all to the end pole.
    The game is to assumed to have 3 poles.
    """

>>> towers_of_hanoi(1, 1, 3)
Move 1 disk from rod 1 to rod 3
>>> towers_of_hanoi(2, 1, 3)
Move 1 disk from rod 1 to rod 2
Move 1 disk from rod 1 to rod 3
Move 1 disk from rod 2 to rod 3
"""