In infinite sequences, the concept of lazy evaluation is crucial. Lazy evaluation is a strategy where expressions are only evaluated when they are needed. This is in contrast to eager evaluation, where all expressions are fully evaluated immediately.

The function `is_better(a, b, a_better, b_better)` demonstrates lazy evaluation:

```python
def is_better(a, b, a_better, b_better):
    if a > b:
        return a_better
    return b_better
```

```python
is_better(foo, bar, fib(1), fib(1000))
```

In lazy (or deferred) evaluation, expressions are only evaluated when they are needed.

Lazy evaluation is native to many languages, such as Haskell.

By contrast, Python is an eager language, evaluating expressions immediately.
**Lazy Evaluation**

We can modify `is_better` to evaluate “lazily” by passing in functions. These functions will then provide the necessary values when—and if—they are called.

```python
def is_better(a, b, a_better_fn, b_better_fn):
    if a > b:
        return a_better_fn()
    return b_better_fn()

is_better(foo, bar, lambda: fib(1), lambda: fib(1000))
```

**Infinite Sequences: Examples**

A sequence is an ordered collection of data values.

There are many kinds of sequences, and all share certain properties.

- **Length**: A sequence has a finite length.
- **Element selection**: A sequence has an element for any non-negative integer less than its length.

**Problem**: How do we represent an infinite sequence in a finite-memory computer?

**Infinite Sequences: Examples**

- Mathematical sequences
  - Prime numbers, Fibonacci sequence, ...
- Internet and cell phone traffic
- Real-time data
  - Instrument measurements, stock prices, weather, social media data, ...

**Problem**: How do we represent an infinite sequence in a finite-memory computer?

**Announcements**

- **Homework 12 due Tuesday, July 31.**
- **Project 4 due Tuesday, August 7.**
  - Partnered project, in two parts.
  - Twelve questions, so please start early!
  - Two extra credit questions.

**Announcements: Midterm 2**

- Scores available on glookup.
  - Average: 39.9, standard deviation: 6.9.
  - Will be handed back in lab today.
- Solutions are available online.
  - Regrade requests due Tuesday, August 7.
- Post-midterm de-stress potluck this week.
  - Food and games.
  - Come and leave when you want.
ANNOUNCEMENTS: FINAL

- Final is Thursday, August 9.
  - Where? 1 Pimentel.
  - When? 6PM to 9PM.
  - How much? All of the material in the course, from June 18 to August 8, will be tested.
- Closed book and closed electronic devices.
- One 8.5” x 11” ‘cheat sheet’ allowed.
- No group portion.
- We will get back to you this week if you have conflicts and have told us. If you haven’t told us yet, please let us know.

REVIEW: RECURSIVE LISTS

\[<1, 2, 3>\]

1 2 3

Third element

Idea: We only construct as much of an infinite sequence as we will need.

We also remember how to construct the rest of the sequence, in case we need to find more values of the infinite sequence.

INFINITE RECURSIVE LISTS

\[<1, 2, 3, \ldots>\]

1 2 3 ...

Observe that we depict only as many values of an infinite sequence as we will need.

STREAMS

Streams are lazily computed recursive lists that represent (potentially infinite) sequences.

Like a recursive list, a stream is a pair: the first element is the first element of the stream, the second element stores how to compute the rest of the stream when needed, and will compute it when asked.
**STREAMS**

```python
class Stream(object):
    def __init__(self, first, compute_rest, empty=False):
        self.first = first
        self._compute_rest = compute_rest
        self.empty = empty
        self._rest = None
        self._computed = False
```

First element of the stream

Is this stream empty?

How to compute the rest of the stream when needed?

Has the rest of this stream already been computed?

**STREAMS: EXAMPLE**

To construct the (finite) stream

$$1, 2, 3$$

we make a Stream object

whose first element is 1, and

whose second element computes the smaller (finite) stream

$$2, 3$$

when called.

```python
Stream(1, lambda: Stream(2, lambda: Stream(3, lambda: Stream.the_empty_stream)))
```

**STREAMS: EXAMPLE**

Notice that the definition

```python
Stream(1, lambda: Stream(2, lambda: Stream(3, lambda: Stream.the_empty_stream)))
```

is similar to the definition of the recursive list <1, 2, 3>:

```python
make_rlist(1, make_rlist(2, make_rlist(3, Rlist.the_empty_rlist)))
```

This is no accident: streams are recursively defined.

The rest of a Stream is also a Stream.

**INFINITE STREAMS: EXAMPLE**

We want to make an infinite stream of only 1s.

$$1, 1, 1, 1, 1, \ldots$$

First element is 1.

The rest is also an infinite stream of only 1s.
**INFINITE STREAMS: EXAMPLE**

```python
def make_one_stream():
    def compute_rest():
        return make_one_stream()
    return Stream(1, compute_rest)
```

First element is 1. Second element is a function that, when called, will return a stream of 1s.

```
make_one_stream() calls make_one_stream recursively, but there is no base case!
This is because we want an infinite stream.
```

**INFINITE STREAMS: EXAMPLE**

```python
def make_one_stream():
    def compute_rest():
        return make_one_stream()
    return Stream(1, compute_rest)
```

```
make_one_stream calls make_one_stream recursively, but there is no base case!
This is because we want an infinite stream.
```

**INFINITE STREAMS: EXAMPLE**

```python
>>> ones = make_one_stream()
>>> ones.first
1
>>> ones.rest.first
1
>>> ones.rest.rest.first
1
```

```
First element is 1. Second element is a function that, when called, will return a stream of 1s.
```

**INFINITE STREAMS: EXAMPLE**

```python
>>> ones = make_one_stream()
>>> ones.rest.first
1
>>> ones.rest.rest.first
1
>>> ones.rest.rest.rest.rest.first
1
```

```
First element is 1. Second element is a function that, when called, will return a stream of 1s.
```

**INFINITE STREAMS: PRACTICE**

Write the function `make_stream_of` that generalizes the stream of 1s to produce a stream of any given number.

```python
>>> fives = make_stream_of(5)
>>> fives.first
5
>>> fives.rest.rest.rest.rest.first
5
```

**INFINITE STREAMS: PRACTICE**

Write the function `make_stream_of` that generalizes the stream of 1s to produce a stream of any given number.

```python
def make_stream_of(n):
    def compute_rest():
        return ______________________
    return ________________________
```

**INFINITE STREAMS: PRACTICE**

Write the function `make_stream_of` that generalizes the stream of 1s to produce a stream of any given number.

```python
def make_stream_of(n):
    def compute_rest():
        return ______________________
    return ________________________
```

**INFINITE STREAMS: PRACTICE**

Write the function `make_stream_of` that generalizes the stream of 1s to produce a stream of any given number.

```python
def make_stream_of(n):
    def compute_rest():
        return ______________________
    return ________________________
```

```python
def make_stream_of(n):
    def compute_rest():
        return ______________________
    return ________________________
```
INFINITE STREAMS: PRACTICE

Write the function integers_starting_from that returns an infinite stream of integers starting from a given number.

```python
def integers_starting_from(n):
    def compute_rest():
        return integers_starting_from(n+1)
    return Stream(n, compute_rest)
```

INFINITE STREAMS: PRACTICE

Write the function integers_starting_from that returns an infinite stream of integers starting from a given number.

```python
def integers_starting_from(n):
    def compute_rest():
        return __________________
    return __________________
```

INFINITE STREAMS: PRACTICE

Write the function integers_starting_from that returns an infinite stream of integers starting from a given number.

```python
def integers_starting_from(n):
    def compute_rest():
        return integers_starting_from(n+1)
    return Stream(n, compute_rest)
```

INFINITE STREAMS

Write a function add_one that adds one to every element of a stream of integers.

```python
def add_one(s):
    if s.empty:
        return Stream.the_empty_stream
    def compute_rest():
        return add_one(s.rest)
    return Stream(s.first + 1, compute_rest)
```

INFINITE STREAMS

Write a function add_one that adds one to every element of a stream of integers.

```python
def add_one(s):
    if s.empty:
        return Stream.the_empty_stream
    def compute_rest():
        return add_one(s.rest)
    return Stream(s.first + 1, compute_rest)
```

We will never reach this base case if the stream is infinite. Why is that okay?
Write a function `map_stream` that applies a given function to each element of a stream, and returns the new stream.

```python
def map_stream(fn, s):
    if s.empty:
        return Stream.the_empty_stream
    def compute_rest():
        return map_stream(fn, s.rest)
    return Stream(fn(s.first), compute_rest)
```

```python
def filter_stream(pred, s):
    if s.empty:
        return Stream.the_empty_stream
    def compute_rest():
        return filter_stream(pred, s.rest)
    if pred(s.first):
        return Stream(s.first, compute_rest)
    return filter_stream(pred, s.rest)
```
INFINITE STREAMS: EXAMPLE

Write a function `add_streams` that takes two infinite streams of numbers, `s1` and `s2`, and produces a new stream of the sum of the numbers at the same positions in the two streams.

```python
>>> s = integers_starting_from(2)
>>> t = integers_starting_from(5)
>>> sum_stream = add_streams(s, t)
>>> sum_stream.rest.first
9
```

---

INTEGER STREAM: ALTERNATE CONSTRUCTION

Now that we have defined `add_streams`, we can construct the stream of non-negative integers, `nonnegative_ints`, in another way:

```
1, 2, 3, 4, 5, ...
= 1, 1, 1, 1, 1, ...
+ 1, 2, 3, 4, ...
```

We can define `nonnegative_ints` in terms of itself!

---

INFINITE STREAMS: EXAMPLE

Write a function `add_streams` that takes two infinite streams of numbers, `s1` and `s2`, and produces a new stream of the sum of the numbers at the same positions in the two streams.

```python
def add_streams(s1, s2):
    def compute_rest():
        return add_streams(s1.rest, s2.rest)
    return Stream(s1.first + s2.first, compute_rest)
```

---

INTEGER STREAM: ALTERNATE CONSTRUCTION

```
nonnegative_ints = \nStream(1,
    lambda: add_streams(ones, nonnegative_ints))
```

---

BREAK


---

ITERATORS

Python natively supports iterators.

`Iterators` are objects that give out one item at a time and save the next item until they are asked: this evaluation is `lazy`.

---
ITERATORS

Python has the following interface for iterators:
• The __iter__ method should return an iterator object.
• The __next__ method should:
  – return a value, or
  – raise a StopIteration when the end of the sequence is reached, and on all subsequent calls.

ITERATORS: EXAMPLE

We want to write an iterator that returns the characters of a word.

```python
>>> jom_chars = Characters('jom')
>>> jom_chars.__next__()
'j'
>>> jom_chars.__next__()
'o'
>>> next(jom_chars)  # This is equivalent to jom_chars.__next__()
'm'
>>> jom_chars.__next__()
Traceback (most recent call last):
  ... StopIteration
```

ITERATORS: EXAMPLE

```python
class Characters:
    def __init__(self, word):
        self.word = word
        self.curr_pos = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.curr_pos >= len(self.word):
            raise StopIteration
        result = self.word[self.curr_pos]
        self.curr_pos += 1
        return result
```

ITERATORS

Why does Python provide this interface? Among other things, so that we can iterate using for-loops.

```python
>>> for char in Characters('jom'):
...     print(char)
...     j
...     o
...     m
```

ITERATORS

What does a for-loop do "under the hood"?

```python
for x in iterable:
    function(x)
```

is equivalent to

```python
iterator = iterable.__iter__()
try:
    while True:
        element = iterator.__next__()
        function(element)
except StopIteration as e:
    pass
```
OTHER LAZY OBJECTS

We have seen a few other lazy objects before, or objects that only produce new values when they are asked.

The objects returned by the built-in map and filter functions are lazy; the range constructor is also lazy.

CONCLUSION

• In lazy evaluation, expressions are not evaluated until they are needed.
• Streams allow us to represent infinite sequences.
• Streams are pairs whose first element is the first element of the stream, and whose second element stores how the rest of the stream can be calculated.
  – This way, only as much of the stream is created as is needed.
• Python has built-in support for iterators, or objects that compute new values only when asked.
• Preview: Generators, logic programming.

EXTRAS: THE SIEVE OF ERATOSTHENES

Eratosthenes (Ἐρατοσθένους) of Cyrene was a Greek mathematician (276 BC – 195 BC) who discovered an algorithm for generating all prime numbers from 2 up to any given limit.

EXTRAS: THE SIEVE OF ERATOSTHENES

def primes(positive_ints):
    # positive_ints is a stream that starts at least at 2.
    def not_div_by_first(num):
        return num % positive_ints.first != 0
    def sieve():
        return primes(filter_stream(not_div_by_first,
                                     positive_ints.rest))
    return Stream(positive_ints.first, sieve)