CS61A Lecture 24
Infinite Sequences

Jom Magrotker
UC Berkeley EECS

July 30, 2012

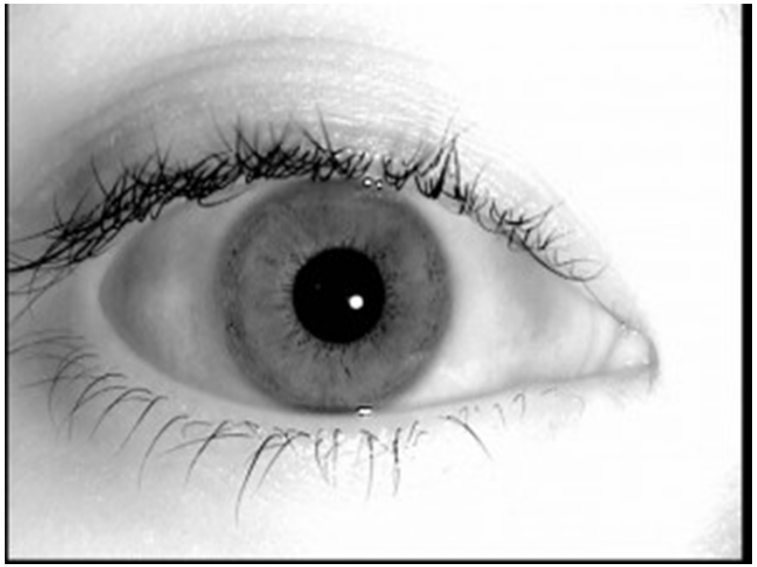
Computer Science in the News

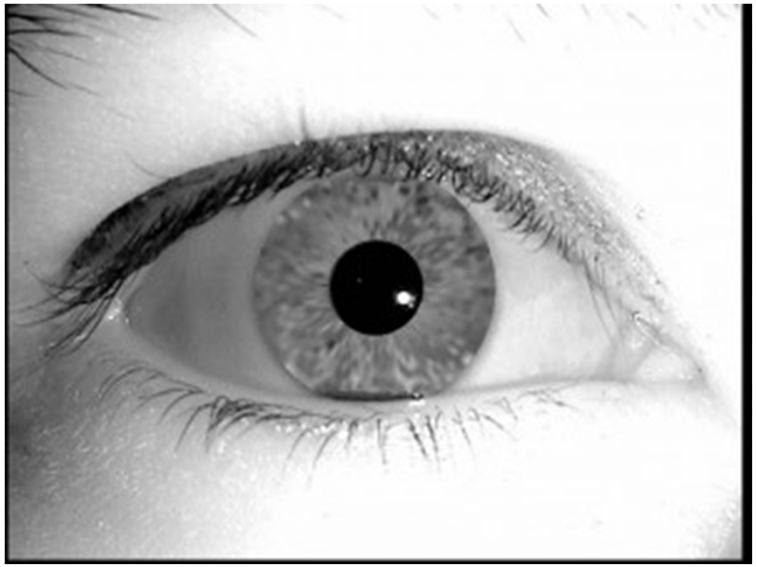
Reverse-Engineered Irises Look So Real, They Fool Eye-Scanners

415 60 42

Tweet +1 Share

By Kim Zetter July 25, 2012 | 6:00 am | Categories: Black Hat Conference, Cybersecurity





http://www.wired.com/threatlevel/2012/07/reverse-engineering-iris-scans/all/

AGENDA

Weeks 1 to 3: Functional programming

Weeks 4 to 5: Object-oriented programming

Next Two Weeks:

- Streams and Lazy Evaluation (*Today*)
- Logic Programming
- Client/Server Programming
- Parallel Programming

LAZY EVALUATION

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

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

Lazy Evaluation

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

Remember that the operand and operators are evaluated first, before the body of the function is executed.

Since the operators are evaluated *before* the body is executed, `fib(1000)` is calculated, even though its value may not be needed.

Lazy Evaluation

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.

```
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*.

What about infinite collections of data, or infinite sequences?

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

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, ...

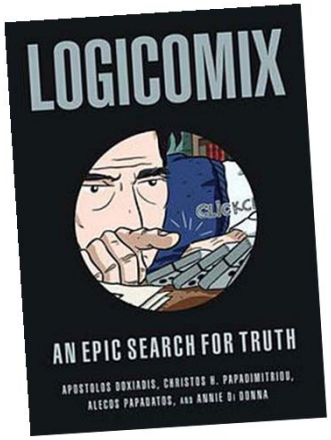
Problem: How do we represent an infinite sequence in a finite-memory computer?

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

- Project 4 contest due **Friday, August 3**.
  - Generate recursive art using Scheme.
  - Prizes awarded in two categories:
    - **Featherweight**: At most 128 words of Scheme.
    - **Heavyweight**: At most 1024 words of Scheme.
  - One question on homework 14 will ask you to vote for your favorite drawing.
  - Three extra credit points.
  - *Prize*: *Logicomix*



http://upload.wikimedia.org/wikipedia/en/thumb/6/60/Logicomix_cover.jpg/250px-Logicomix_cover.jpg

Announcements: Midterm 2

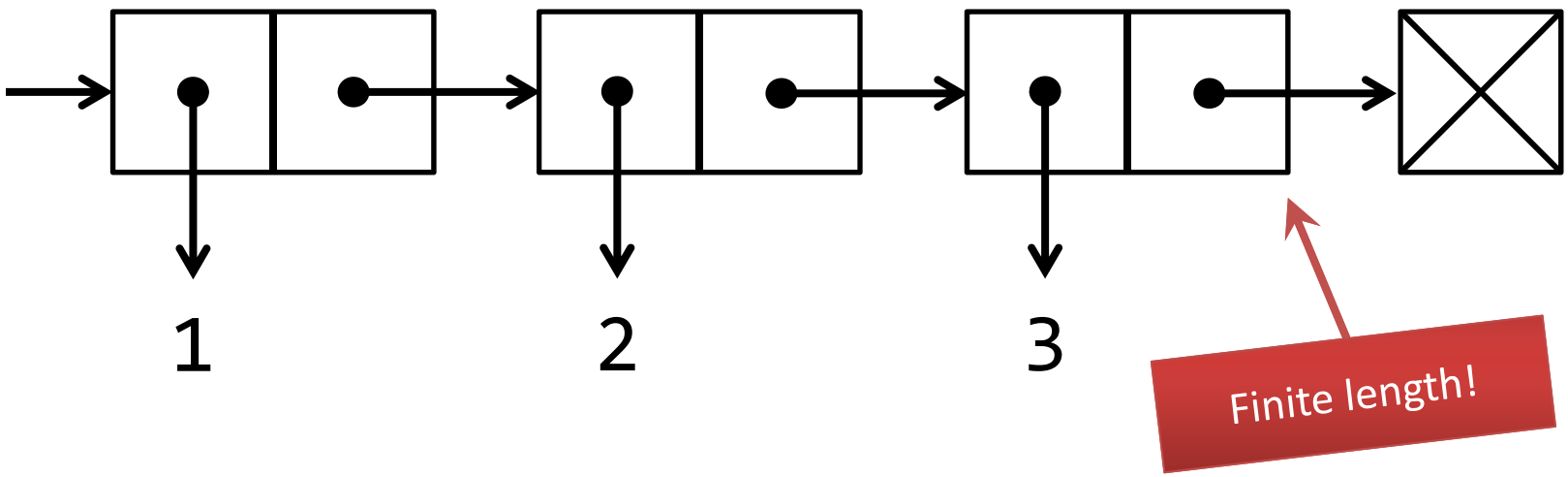
- Scores available on `glookup`.
  - Average: 39.9, standard deviation: 6.9.
  - Will be handed back in lab today.
- Solutions are available online.
  - Regrade requests due **Tuesday, August 7**.
- Post-midterm de-stress potluck this week.
  - Food and games.
  - Come and leave when you want.

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*.

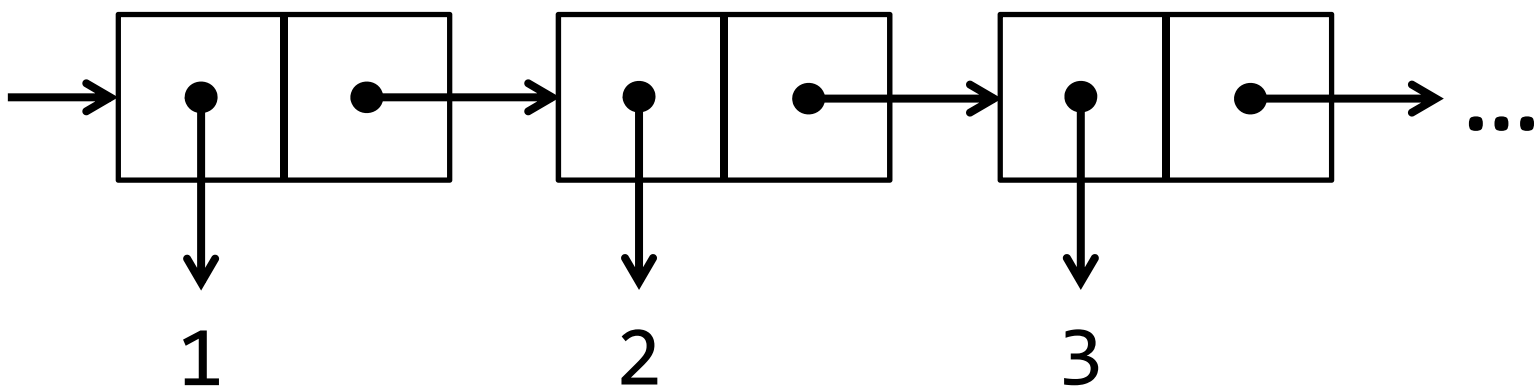
Review: Recursive Lists

<1, 2, 3>



Infinite Recursive Lists

<1, 2, 3, ...>



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

Infinite Recursive Lists

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.

STREAMS

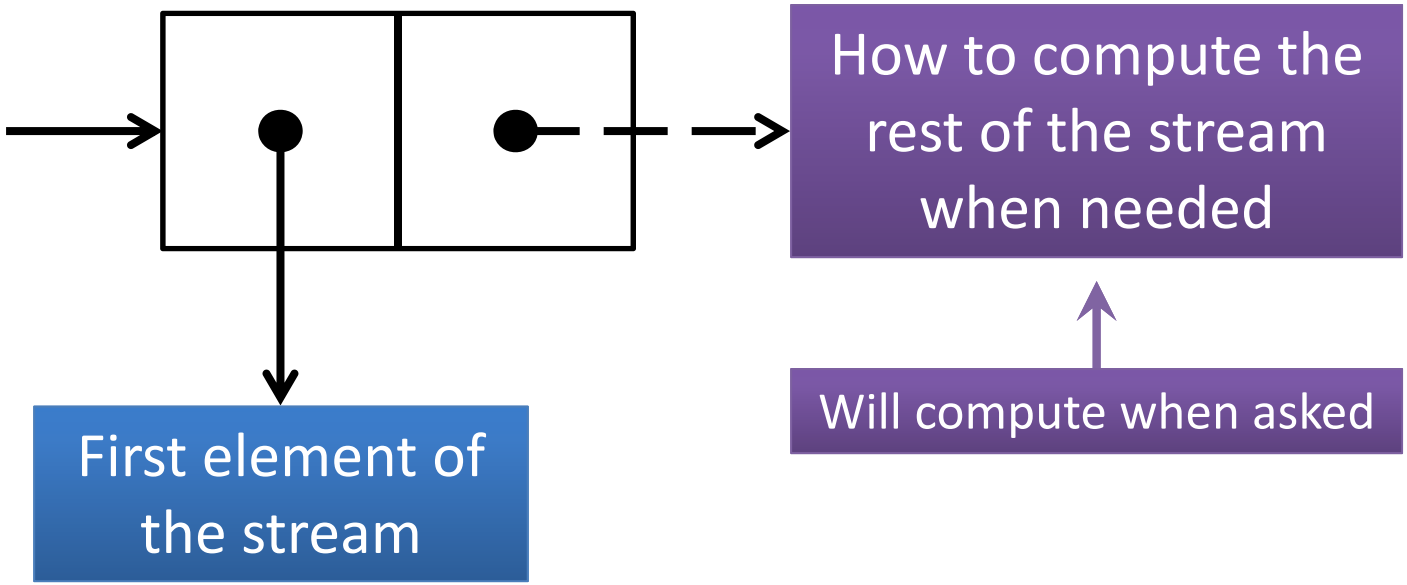
Streams are *lazily computed* recursive lists that represent (potentially infinite) sequences.

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


How to compute the rest of the stream when needed
Will compute when asked
First element of the stream

STREAMS

```
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

How to compute the rest of the stream when needed

Is this stream empty?

Has the rest of this stream already been computed?

STREAMS

```
class Stream:
    ...
    @property
    def rest(self):
        assert not self.empty, \
        'Empty streams have no rest.'
        if not self._computed:
            self._rest = self._compute_rest()
            self._computed = True
        return self._rest

Stream.the_empty_stream = Stream(None, None, True)
```

If the rest of the stream has not been computed yet...

... compute it and remember the result.

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.

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

STREAMS: EXAMPLE

Notice that the definition

```
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>`:

```
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`.

STREAMS: EXAMPLE

```
>>> s = Stream(1,
               lambda:
               Stream(2, lambda:
                         Stream(3,
                                lambda:
                                Stream.the_empty_stream)))
>>> s.first
1
>>> s.rest.first
2
>>> s.rest.rest.first
3
>>> s.rest.rest.rest
<empty stream>
```

The interaction with a stream is very similar to that with a recursive list.

Infinite Streams: Example

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

```
1, 1, 1, 1, 1, ...
```

First element is 1.

The rest is *also* an infinite stream of only 1s.

Infinite Streams: Example

```
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.

Infinite Streams: Example

```
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

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

Infinite Streams: Practice

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

```
>>> 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.

```
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.

```
def make_stream_of(n):
    def compute_rest():
        return make_stream_of(n)
    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.

```
>>> positive_ints = \
    integers_starting_from(0)
>>> positive_ints.first
0
>>> positive_ints.rest.rest.rest.first
3
```

INFINITE STREAMS: PRACTICE

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

```
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.

```
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.

```
>>> s = make_stream_of(3)
>>> t = add_one(s)
>>> t.first
4
```

Infinite Streams

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

```
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)
```

If the stream is empty, return the empty stream.

The rest of the new stream…

… is one added to the rest of the original stream.

The first of the new stream is one more than the first of the original stream.

Infinite Streams

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

```
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?

Infinite Streams: Practice

Write a function `map_stream` that applies a given function to each element of a stream, and returns the new stream.

```
>>> s = integers_starting_from(1)
>>> t = map_stream(lambda x: x*x, s)
>>> t.rest.first
4
```

INFINITE STREAMS: PRACTICE

Write a function `map_stream` that applies a given function to each element of a stream, and returns the new stream.

```
def map_stream(fn, s):
    if s.empty:
        return ________________________
    def compute_rest():
        return ________________________
    return Stream(____________, compute_rest)
```

INFINITE STREAMS: PRACTICE

Write a function `map_stream` that applies a given function to each element of a stream, and returns the new stream.

```
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)
```

INFINITE STREAMS: PRACTICE

Write a function `filter_stream`, which takes in a predicate function and a stream, and returns a new stream of values that satisfy the predicate function.

```
>>> s = integers_starting_from(2)
>>> t = filter_stream(lambda x: x%2 == 0, s)
>>> t.rest.first
4
```

INFINITE STREAMS: PRACTICE

```
def filter_stream(pred, s):
    if s.empty:
        return ________________________
    def compute_rest():
        return ____________________________
    if _____________:
        return Stream(_______, compute_rest)
    return ____________________________
```

Infinite Streams: Practice

```
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.

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

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.

```
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

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, ...
```

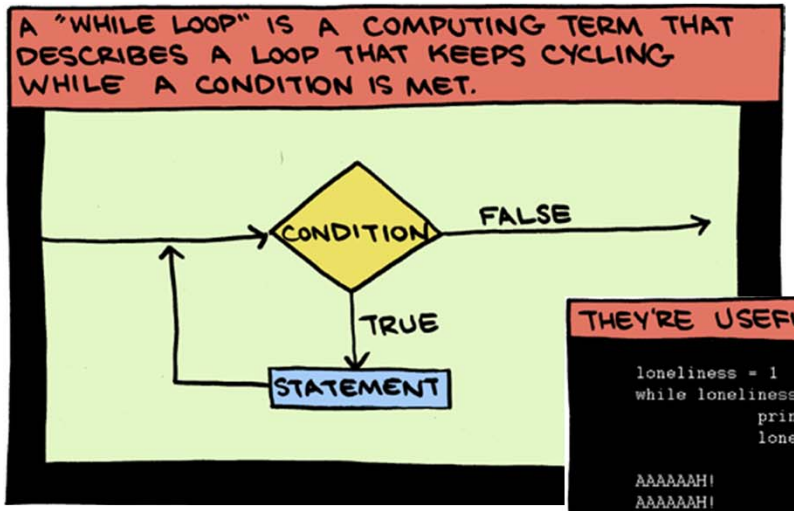
We can define `nonnegative_ints` in terms of itself!

Integer Stream: Alternate Construction

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

Protip: As with recursion, when coding with streams, it helps if you "trust" that the stream is available for you already. How would you then use this information?

BREAK



http://www.smbc-comics.com/index.php?db=comics&id=2656

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.

```
>>> jom_chars = Characters('jom')
>>> jom_chars.__next__()
'j'
>>> jom_chars.__next__()
'o'
>>> next(jom_chars)
'm'
>>> jom_chars.__next__()
Traceback (most recent call last):
    ...
StopIteration
```

The built-in function `next` calls the `__next__` method of its argument.

ITERATORS: EXAMPLE

```
class Characters:
    def __init__(self, word):
        self.word = word
        self.curr_pos = 0

    def __iter__(self):
        return self
```

Remember the position of the character to return.

ITERATORS: EXAMPLE

```
class Characters:
    ...
    def __next__(self):
        if self.curr_pos >= len(self.word):
            raise StopIteration

        result = self.word[self.curr_pos]
        self.curr_pos += 1
        return result
```

If we are done with the word, we raise a `StopIteration`.

Otherwise, return the current character, and "move forward".

ITERATORS

Why does Python provide this interface?

Among other things, so that we can iterate using `for`-loops.

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

ITERATORS

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

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

is equivalent to

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

Create an iterator.

Try to get an element from the iterator.

Apply the function on the element.

If we could not get an element, we *catch* the `StopIteration` exception and do not apply the function.

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.

Other Lazy Objects

```
>>> my_squares = map(square,
                     (4, 5, 6, 7))
>>> my_squares
<map object at 0x...>
>>> my_squares.__next__()
16
>>> next(my_squares)
25
```

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.

http://upload.wikimedia.org/wikipedia/commons/thumb/a/a2/Portrait_of_Eratosthenes.png/220px-Portrait_of_Eratosthenes.png

Extras: The Sieve of Eratosthenes

1. First, remove all the numbers that are multiples of 2, except for 2 itself.
2. Then, remove all the numbers that are multiples of the next non-removed number, except for that number itself.
3. Repeat step 2 until no more numbers can be removed.

Demo:
http://en.wikipedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif

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)
```