CS 61A
Discussion 10

Iterators, Generators, Streams

Raymond Chan
Discussion 121
UC Berkeley

Agenda

- Announcements
- Iterators
- Generators
- Streams

Announcements

- Homework 7 due Friday (4/15) (Homework party tonight)
- Lab 11 due Friday
- Scheme Project due 4/23
  - Submit Part 1 by Monday (4/18) for 1 EC
- Maps Composition Revision due 4/15

Iterators

- An iterator is an object that tracks the position in a sequence of values.
- It returns elements one at a time.
- Can only go through the elements once.

Iterators

```
class Natruals():
  def __init__(self, end):
    self.current = 0
    self.end = end

  def __next__(self):
    if self.current > self.end:
      raise StopIteration
    result = self.current
    self.current += 1
    return result

def __iter__(self):
   return self
```

Iterators

- **__next__(self)**: checks for the next value in the sequence.
    - If there are values left, it computes and returns the next element.
    - Keeps track of the current position/state.
    - Raises a **StopIteration** exception to signal the end of the sequence.
    - Since each call can return different values, **__next__** is a non-pure function.

Iterators

- **__iter__(self)** always returns an iterator.

- An iterator is a class that has implemented both **__next__** and **__iter__**.

- The **__iter__** of an iterator returns self.

Iterators

- Iterables are sequences that can be iterated over.
  - Examples: lists, tuples, strings, dictionaries.
- Has an __**iter**__ method that returns a *new* iterator.
  - Allow us to iterate over a sequence many times.
  - Iterators don’t reset

Iterators

↓

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
```

Iterators

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
>>> next(iter_lst)
1
```

Iterators

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
>>> next(iter_lst)
1
>>> next(iter_lst)
2
```

Iterators

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
>>> next(iter_lst)
1
>>> next(iter_lst)
2
>>> next(iter_lst)
3
```

Iterators

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
>>> next(iter_lst)
1
>>> next(iter_lst)
2
>>> next(iter_lst)
3
```

```
>>> next(iter_lst)
4
```

Iterators

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
>>> next(iter_lst)
1
>>> next(iter_lst)
2
>>> next(iter_lst)
3
>>> next(iter_lst)
4
>>> next(iter_lst)
5
```

Iterators

[1, 2, 3, 4, 5]

```
>>> lst = [1, 2, 3, 4, 5]
>>> iter_lst = iter(lst)
>>> next(iter_lst)
1
>>> next(iter_lst)
2
>>> next(iter_lst)
3
>>> next(iter_lst)
4
>>> next(iter_lst)
5
>>> next(iter_lst)
StopIteration Error
```

Iterators

- A **for** loop calls iter on the iterable and continuously calls next on the iterator until a StopIteration Exception is caught.

```
for elem in <iterable>:
    ..........
```

Iterators

iterable	iterator
__iter__(self)	__iter__(self)
	__next__(self

Generators

- A generator function uses a yield statement instead of return.
- It returns can generator function that can be is can be iterated over.
- Each time we call **next** on the generator object, we executed until **yield**.
- At yield, we return the statement and *pauses* frame.
- The next time we call **next**, we start from the line directly beneath yield

Generators

```
def generate_naturals():
  current = 0
  while True:
    yield current
    current += 1
```

```
>>> gen = generate_natruals()
>>> gen
<generator object gen at …>
```

Generators

```
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
```

```
>>> gen = generate_natruals()
>>> gen
<generator object gen at …>
>>> next(gen)
0
```

Generators

```
def generate_naturals():
    current = 0
    while True:
        yield current
    ⟶  current += 1
```

```
>>> gen = generate_natruals()
>>> gen
<generator object gen at …>
>>> next(gen)
0
>>> next(gen)
```

Generators

```
def generate_naturals():
    current = 0
→   while True:
        yield current
        current += 1
```

```
>>> gen = generate_natruals()
>>> gen
<generator object gen at …>
>>> next(gen)
0
>>> next(gen)
```

Generators

```
def generate_naturals():
    current = 0
    while True:
→       yield current
        current += 1
```

```
>>> gen = generate_natruals()
>>> gen
<generator object gen at …>
>>> next(gen)
0
>>> next(gen)
1
```

Generators

- Since we can call next on generator objects, we can create iterators with no __**next**__ method.

- The __**iter**__ method would have to return a generator object.

Streams (Scheme)

- Iterators and generators are *lazy* and can potentially represent infinite sequences.
- We only compute the next value when we ask for it.
- Scheme Lists cannot be infinite.

Streams

- The the second argument to **cons** is always evaluated.

```
> (define (naturals n)
      (cons n (naturals (+ n 1))))
> Maximum Recursion Depth Reached
```

Streams

- Streams are lazy Scheme Lists.
- The rest of the list is not evaluated until you ask for it.
- Once you have asked for it once, it will save (memoize) the value so that it will not be evaluated again.

Streams

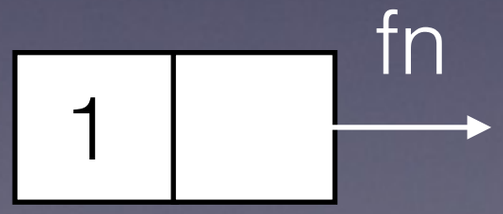
- **cons-stream** creates a pair where the second is a stream.
- **nil** is an empty stream.
- **car** returns the first element.
- **cdr-stream** *computes* and *returns* the rest of the stream.
- **cdr** will not calculate the next value.

Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
```

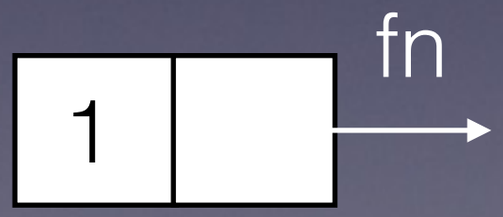
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
```



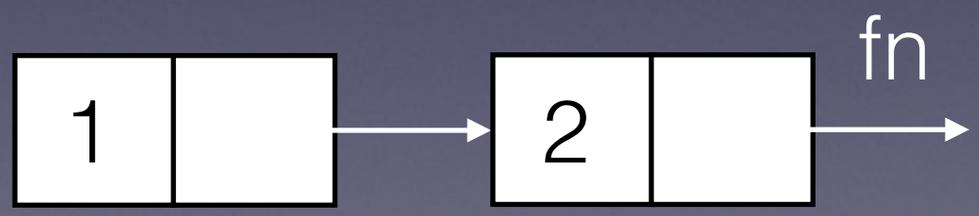
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
```



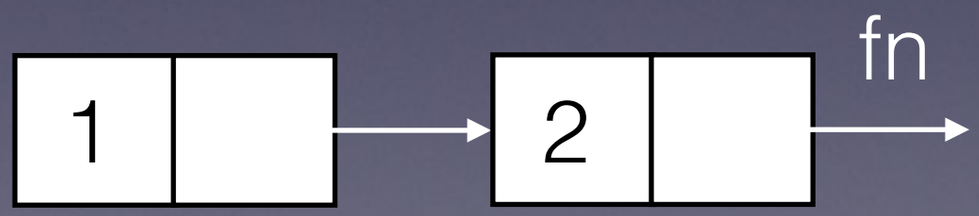
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
```



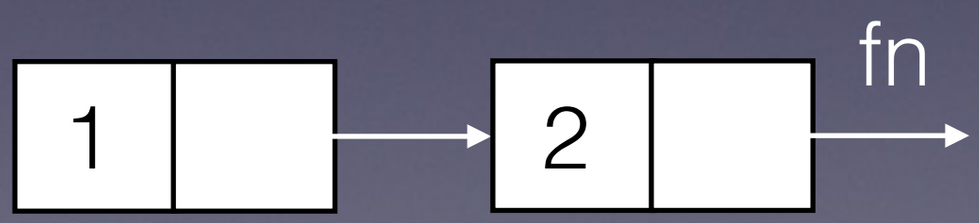
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
```



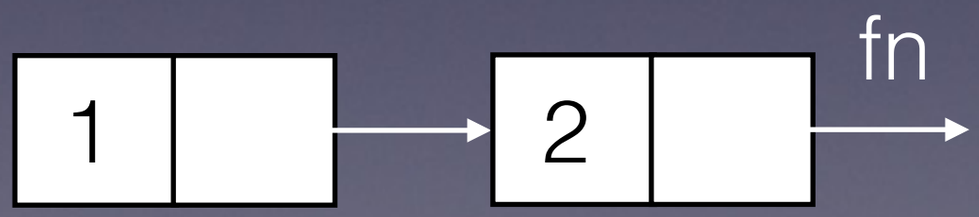
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
```



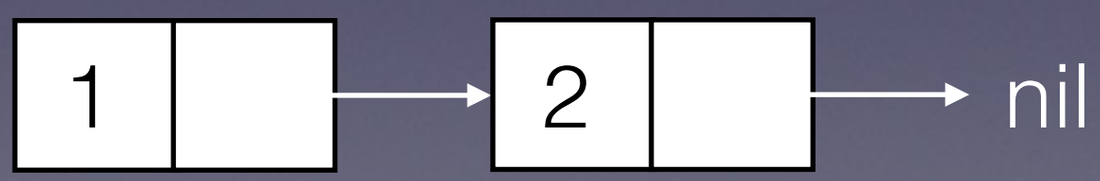
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
```



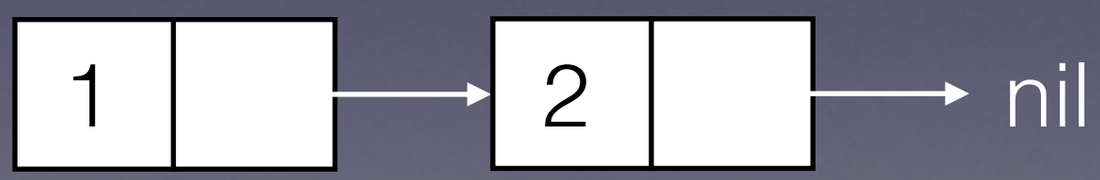
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
()
```



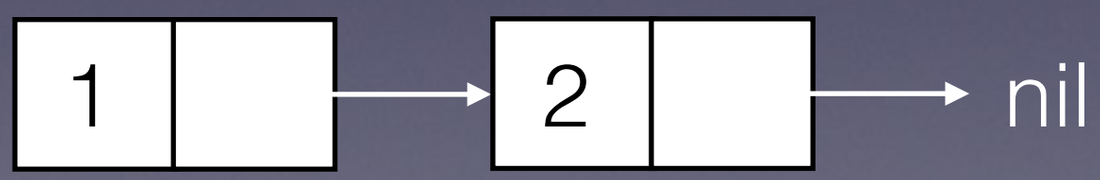
Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
()
> (cdr s)
```



Streams

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
()
> (cdr s)
#[promised (forced)]
```



Streams (Python)

- Streams in Python are lazy linked lists.

- There rest of the linked lists is not computed yet until we need it.

Streams (Python)

```
class Stream:
  class empty:
    def __repr__(self):
      return 'Stream.empty'

  empty = empty()

  def __init__(self, first, compute_rest=lambda: Stream.empty):
    assert callable(compute_rest), 'compute_rest must be callable.'
    self.first = first
    self._compute_rest = compute_rest

@property
def rest(self):
  """Return the rest of the stream, computing it if necessary."""
  if self._compute_rest is not None:
      self._rest = self._compute_rest()
      self._compute_rest = None
  return self._rest

def __repr__(self):
  return 'Stream({0}, <...>)'.format(repr(self.first))
```

Streams (Python)

- **__init__(self, first, compute_rest)**

- **compute_rest** is a function with 0 parameters that returns the rest of the stream.

- By default it is a lambda function that returns **Stream.empty**.

Streams (Python)

- Let **s** be a Stream instance.
- At initialization, **self._compute_rest** is assigned to the function that we pass in.

```
def __init__(self, first, compute_rest=lambda: Stream.empty):
  assert callable(compute_rest), 'compute_rest must be callable.'
  self.first = first
  self._compute_rest = compute_rest
```

Streams (Python)

- When we call **s.rest** the first time, we will calculate the rest of the stream by calling **self._compute_rest()**.

```
@property
def rest(self):
  """Return the rest of the stream, computing it if necessary."""
  if self._compute_rest is not None:
      self._rest = self._compute_rest()
      self._compute_rest = None
  return self._rest
```

Streams (Python)

- For all subsequent calls to **s.rest**, we can just return **self._rest** without calculating the value again.

```
@property
def rest(self):
  """Return the rest of the stream, computing it if necessary."""
  if self._compute_rest is not None:
      self._rest = self._compute_rest()
      self._compute_rest = None
  return self._rest
```

Recap

- Iterators goes over the elements of a sequence one at a time.
- Generators return generator objects that returns at **yield** and passes the frame.
- Streams are lists such that the rest of the list is not calculated until we need it.