CS61A Lecture 25
Delayed Sequences

Jom Magrotker
UC Berkeley EECS
July 31, 2012

TODAY

• Review Iterators
• Review Streams
• Another way of defining a sequence of values using delayed evaluation: Generators

REVIEW: 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.

REVIEW: 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.

REVIEW: 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
```

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.
ANNOUNCEMENTS

• Homework 12 due Today.
• Homework 13 due Friday.
  – Out later today.
  – Includes Py, Streams, Iterators, and Generators
  – Also includes the Project 4 contest.
• 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.
  – Extra credit point to the top 3 in each category.
  – Prize: Logicomix

ANNOUNCEMENTS: MIDTERM 2

• Scores available on glookup.
  – Average: 39.9, standard deviation: 6.9.
• 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.
  – When? 6PM to 9PM.
  – How much? All of the material in the course, from June 18 to August 8, will be tested.
  – 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: 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.
REVIEW: 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

Second element of the stream

How to compute the rest of the stream when needed

Has the rest of this stream already been computed?

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

REVIEW: STREAMS

```
@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
```

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

compute it and remember the result.

PRACTICE: STREAMS

What are the first 9 elements of the stream `s` defined below?

```python
>>> s = Stream(1, lambda: add_streams(s, s))
```

Value 1 2 4 8 16 32 64 128 256

Position 0 1 2 3 4 5 6 7 8

Computation S + S

EXAMPLE: STREAMS

Say I wanted to take two input streams and return a stream which alternates between items from each of the input streams. Let’s call this `interleave(s1, s2)`.

The way it should behave is like this:

```python
>>> ones = Stream(1, lambda: ones)
>>> ints = Stream(1,
    lambda: add_streams(ones, ints))
>>> mixed = interleave(ones, ints)
>>> show_stream(mixed)
1, 1, 2, 1, 3, 1, 4, 1, 5, ...
```
**EXAMPLE: STREAMS**

Say I wanted to take two input streams and return a stream which alternates between items from each of the input streams. Let’s call this `interleave(s1, s2)`.

How do we implement it?

```python
def interleave(s1, s2):
    return Stream(s1.first, lambda: interleave(s2, s1.rest))
```

The result of interleaving `s1` and `s2` is...

- the first item of `s1`...
- followed by items of `s2`...
- interleaved with the remaining items of `s1`.

**PRACTICE: STREAMS**

Say I wanted to create `ab_stream`, which is a stream that contains all strings created using some number of “a” and “b” strings, including the empty string “”. Using `stream_map` and `interleave`, create `ab_stream`.

```python
add_a = lambda x: x + “a”
add_b = lambda x: x + “b”
ab_stream = Stream(“”, lambda: interleave(stream_map(add_a, ab_stream), stream_map(add_b, ab_stream)))
```

Start with the empty string...

- Alternate between...
- items where we add an “a”...
- items where we add a “b”...

**BREAK**

Metal lessons from Pythons

#428 Don’t be self-centered.

>>> ‘I’ > ‘you’
False


**GENERATORS**

Generators are an elegant and concise way to define an iterator. Example: Make an iterator that returns the sequence [0], [0, 1], [0, 1, 2], ...

```python
def triangle():
    lst, cur = [], 0
    while True:
        yield lst + [cur]
        lst, cur = lst + [cur], cur + 1
    for lst in triangle():
        print(lst)
```

...wait, what?

Generators use the `yield` keyword.

When a function definition includes a yield statement, Python knows that a call to the function should return a generator object.
GENERATORS: EXPLAINED

To produce each item, Python starts executing code from the body of the function.

If a yield statement is reached, Python stops the function there and yields the value as the next value in the sequence. When the generator is asked for another value, the function resumes executing on the line after where it left off.

When the function returns, then there are no more values “in” the generator.

ANOTHER EXAMPLE: GENERATORS

Suppose I wanted to make a new generator based on another generator. Let’s write generator_map, which takes a function and a generator and produces the generator which yields the results of applying the function to each item yielded by the input generator.

def ints():
    cur = 1
    while True:
        yield cur
        cur += 1

for item in generator_map(lambda x: 2 * x, ints()):
    print(item)
    2
    4
    6...

PRACTICE: GENERATORS

Write the generator generator_filter, which takes a predicate, pred, and a generator, gen, and creates a generator which yields the items yielded by gen for which pred returns True.

>>> odd = lambda x: x % 2 == 1
>>> for item in generator_filter(odd, ints()):
...     print(item)
... 1
... 3
... 5...

GENERATORS

>>> def triangle():
...     lst, cur = [], 0
...     while True:
...         yield lst + [cur]
...         lst, cur = lst + [cur], cur + 1
... >>> for lst in triangle():
...         print(lst)
... [0]
... [0, 1]
... [0, 1, 2]

PRACTICE: GENERATORS

Write the generator generator_filter, which takes a predicate, pred, and a generator, gen, and creates a generator which yields the items yielded by gen for which pred returns True.

def generator_filter(pred, gen):
    for item in gen:
        if pred(item):
            yield item
CONCLUSION

• In lazy evaluation, expressions are not evaluated until they are needed.
• Python has built-in support for iterators, or objects that give back a new value each time you ask.
• Streams allow us to represent infinite sequences using delayed evaluation.
• 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.
• Generators are a convenient and elegant way to define new iterators using yield statements.
• Preview: Declarative Programming