61A LECTURE 24 – STREAMS, GENERATORS

Steven Tang and Eric Tzeng

August 5, 2013

Our Sequence Abstraction

Recall our previous sequence interface:

- A sequence has a finite, known length
- A sequence allows element selection for any element

In the cases we’ve seen so far, satisfying the sequence interface requires storing the entire sequence in a computer's memory

Problems?

- Infinite sequences - primes, positive integers
- Really large sequences - all Twitter tweets, votes in a presidential election

Implicit Sequences

- We compute each of the elements on demand
- Don’t explicitly store each element
- Called an **implicit sequence**

A Python Example

Example: The `range` class represents a regular sequence of integers

- The range is represented by three values: *start*, *end*, and *step*
- The length and elements are computed on demand
- Constant space for arbitrarily long sequences

$$length = max\left(\left\lceil \frac{end - start}{step} \right\rceil, 0\right)$$

$$elem(k) = start + k \cdot step \qquad (\text{for } k \in [0, length))$$

A Range Class

```
class Range(object):
    def __init__(self, start, end=None, step=1):
        if end is None:
            start, end = 0, start
        self.start = start
        self.end = end
        self.step = step

    def __len__(self):
        return max(0, ceil((self.end - self.start) /
                                   self.step))

    def __getitem__(self, k):
        if k < 0:
            k = len(self) + k
        if k < 0 or k >= len(self):
            raise IndexError('index out of range')
        return self.start + k * self.step
```

The Iterator Interface

An iterator is an object that can provide the next element of a (possibly implicit) sequence

The iterator interface has two methods:

- **`__iter__(self)`** returns an equivalent iterator
- **`__next__(self)`** returns the next element in the sequence
  - If no next, raises **`StopIteration`** exception

There are also built in functions **`next`** and **`iter`** that call the corresponding method on their argument.

RangeIter

```python
class RangeIter(object):
    def __init__(self, start, end, step):
        self.current = start
        self.end = end
        self.step = step
        self.sign = 1 if step > 0 else -1

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

    def __iter__(self):
        return self
```

For now, always returns self!
(Why do we have this then…?)

Fibonacci

```
class FibIter(object):
    def __init__(self):
        self.prev = -1
        self.current = 1

    def __next__(self):
        self.prev, self.current = (self.current,
                                   self.prev + self.current)
        return self.current

    def __iter__(self):
        return self
```

The For Statement

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header **<expression>**, which yields an iterable object.
2. For each element in that sequence, in order:
    A. Bind **<name>** to that element in the first frame of the current environment
    B. Execute the **<suite>**

An iterable object has a method **__iter__** that returns an iterator

```
>>> nums, sum = [1, 2, 3], 0
>>> for item in nums:
        sum += item
>>> sum
6
```

```
>>> nums, sum = [1, 2, 3], 0
>>> items = nums.__iter__()
>>> try:
        while True:
            item = items.__next__()
            sum += item
    except StopIteration:
        pass
>>> sum
6
```

Generators and Generator Functions

Generators:

- An iterator backed by a function, called a **generator function**

Generator Functions:

- A function that returns a generator
- Can tell by looking for the **yield** keyword
- Another example of a *continuation*

A simple generator

```
def ones_generator():
    while True:
        yield 1
```

- The `yield` keyword is what marks this as a generator function
- Calling this function won't do anything besides return a generator object (an iterator)
- Each time we ask for a value from the iterator, it runs the function until it reaches a `yield` statement and gives whatever value was yielded
- The next time we ask for a value, it picks up where it left off
- This iterator will keep giving you ones forever!

Iterating over an Rlist

We can iterate over a sequence even if it has no **`__iter__`** method

Python uses **`__getitem__`** instead, iterating until **`IndexError`** is raised

```
class Rlist(object):
    def __init__(self, first, rest=empty):
        self.first, self.rest = first, rest

    def __getitem__(self, k):
        if k == 0:
            return self.first
        if self.rest is Rlist.empty:
            raise IndexError('index out of range')
        return self.rest[k - 1]
```

How long does it take to iterate over an **`Rlist`** of *n* items? $\Theta(n^2)$

Iterating over an Rlist

We can define an iterator for **Rlist**s using a generator function

```
class Rlist(object):
    def __init__(self, first, rest=empty):
        self.first, self.rest = first, rest

    def __getitem__(self, k):
        if k == 0:
            return self.first
        if self.rest is Rlist.empty:
            raise IndexError('index out of range')
        return self.rest[k - 1]

    def __iter__(self):
        current = self
        while current is not Rlist.empty:
            yield current.first
            current = current.rest
```

Generator method (returns an iterator)

How long does it take to iterate over an **Rlist** of *n* items? $\Theta(n)$

Fibonacci Generator

A generator function that lazily computes the Fibonacci sequence:

```
def fib_generator():
    yield 0
    prev, current = 0, 1
    while True:
        yield current
        prev, current = current, prev + current
```

A generator expression is like a list comprehension, but it produces a lazy generator rather than a list:

```
double_fibs = (fib * 2 for fib in fib_generator())
```

Generator Semantics

```
def fib_generator():
    yield 0
    prev, current = 0, 1
    while True:
        yield current
        prev, current = current, prev + current
```

Calling a generator function returns an iterator that stores a frame for the function, its body, and the current location in the body

Calling **`next`** on the iterator resumes execution of the body at the current location, until a **`yield`** is reached

The yielded value is returned by **`next`**, and execution of the body is halted until the next call to **`next`**

When execution reaches the end of the body, a **`StopIteration`** is raised

Map and Filter

```
def map_gen(fn, iterable):
    iterator = iter(iterable)
    while True:
        yield fn(next(iterator))

def filter_gen(fn, iterable):
    iterator = iter(iterable)
    while True:
        item = next(iterator)
        if fn(item):
            yield item
```

Why don't we need to check if the iterator still has elements?

Bitstring Generator

```
from itertools import product

def bitstrings():
    """Generate bitstrings in order of increasing
    size.

    >>> bs = bitstrings()
    >>> [next(bs) for _ in range(0, 8)]
    ['', '0', '1', '00', '01', '10', '11', '000']
    """
    size = 0
    while True:
        tuples = product(('0', '1'), repeat=size)
        for elem in tuples:
            yield ''.join(elem)
        size += 1
```

Break

Infinite Sequences with Selection

We now have implicit sequences in the form of iterators

Such sequences may be infinite, and they might be lazily evaluated

What if we want to support element selection on infinite sequences?

Let's try creating a **`list`** out of an infinite sequence

```
>>> list(fib_generator())
```

Oops! Infinite loop!

A **`list`** provides immediate access to all elements

But an **`Rlist`** only provides immediate access to its *first* element

The rest can be computed lazily!

Streams

A stream is a recursive list with an *explicit* first element and a *lazily computed* rest-of-the-list

```
class Stream(Rlist):
    """A lazily computed recursive list."""
    def __init__(self, first,
                 compute_rest=lambda: Stream.empty):
        assert callable(compute_rest)
        self.first = first
        self._compute_rest = compute_rest
        self._rest = None

    @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
```

"Please don't reference directly"

Integer Streams

An integer stream is a stream of consecutive integers

An integer stream starting at *k* consists of *k* and a function that returns the integer stream starting at *k+1*

```
def integer_stream(first=1):
    """Return a stream of consecutive integers, starting
    with first.

    >>> s = integer_stream(3)
    >>> s.first
    3
    >>> s.rest.first
    4
    """
    def compute_rest():
        return integer_stream(first+1)
    return Stream(first, compute_rest)
```

Mapping a Function over a Stream

Mapping a function over a stream applies a function only to the first element right away

The rest is computed lazily

```
def map_stream(fn, s):
    """Map fn over the elements of stream s."""
    if s is Stream.empty:
        return s

    def compute_rest():
        return map_stream(fn, s.rest)

    return Stream(fn(s.first), compute_rest)
```

This body is not executed until **compute_rest** is called

Not called yet

Filtering a Stream

When filtering a stream, processing continues until an element is kept in the output

```
def filter_stream(fn, s):
    """Filter stream s with predicate function fn."""
    if s is Stream.empty:
        return s

    def compute_rest():
        return filter_stream(fn, s.rest)

    if fn(s.first):
        return Stream(s.first, compute_rest)
    else:
        return compute_rest()
```

Find an element in the rest of the stream

A Stream of Primes

The stream of integers not divisible by any *k* <= *n* is:

- The stream of integers not divisible by any *k* < *n*,
- Filtered to remove any element divisible by *n*
- This recurrence is called the *Sieve of Eratosthenes*

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

```
def primes(istream):
    """Return a stream of primes, given a stream of
    consecutive integers."""
    def compute_rest():
        not_divisible = lambda x: x % istream.first != 0
        return primes(filter_stream(not_divisible,
                                    istream.rest))
    return Stream(istream.first, compute_rest)
```

Try it

- Write a function `add_streams` that takes two streams and returns a new stream formed by summing corresponding elements in the argument streams.
- Bonus: see if you can use `add_streams` to define to define the Fibonacci stream!