Iterators and Iterables

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
  * Recite prime numbers.
* __next__(self) returns the next element in the sequence
  * Next prime, etc.
  * If no next, raises StopIteration exception

An iterable is a container that provides an __iter__ method
* __iter__(self) returns an iterator over the elements in the container.

Generator Functions

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

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.

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? \(O(n^2)\)

Iterating over an Rlist

We can define an iterator for Rlists 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

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

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

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()

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

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(istream.filter(not_divisible, istream.rest))
    return Stream(istream.first, compute_rest)
Given a stream of 1-argument functions, we can construct a function that is not in the stream, assuming that all functions in the stream terminate.

```python
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[f] T T T T T T T . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
<tr>
<td>[f] F F F F F F F . . .</td>
<td>. . .</td>
</tr>
</tbody>
</table>