Lecture 29: Generators, Streams, and Lazy Evaluation

- Some of the most interesting real-world problems in computer science center around sequential data.
  - DNA sequences.
  - Web and cell-phone traffic streams.
  - The social data stream.
  - Series of measurements from instruments on a robot.
  - Stock prices, weather patterns.
- ...which perhaps is why Python (and other languages) devote a lot of attention to them.

Classes of sequences

- We started with tuples and lists, which are *collections of data* that are computed before being used.
- Constructs such as **for** first turn these into *iterators*, which are *functions* that compute values as they are asked for.
- There's no particular reason why these data have to have been computed beforehand.
- For example, in Lecture 17, we had a type Range, which was like Python's type range:

```
class Range:
    def __init__(self, low, high):
        self._low = low
        self._high = high
    def __iter__(self):
        return RangeIter(self)
```

- A Range is a sequence (low to high), whose individual members are not stored, and are produced (by RangeIter) only when needed.

Generators

- Iterators are objects whose `__next__` method produces values.
- Each call to `__next__` completes before producing a value, so the iterator object must explicitly store the state needed to figure out where in the sequence one is. This can be annoying.
- Python also provides an entirely different mechanism for this purpose: the *generator.*
- A generator is a kind of *suspendable function* or *coroutine*.
- A special statement, **yield** E, means "stop executing this function for the time being, and hand the value E back to whoever called you."
- When the generator function is next called, it picks up where it left off.

Example: Range redux

- An alternative definition of Range:

```
class Range:
    def __init__(self, low, high): self._low = low; self._high = high
    def __iter__(self): return self._generate()
    def _generate(self):
        i = self._low
        while i < self._high:
            yield i
            i += 1
# To use:
for x in Range(0, 10):
    print(x)
```

- Calling self._generate() creates a generator (any function containing a **yield** produces a generator when called).
- Calling __next__ or send on the generator then resumes execution (the first time at the beginning) until getting to **yield**, which tells what value to return.
- If instead control reaches the end, the caller gets a StopIteration

Generators Within Generators

- In Lectures #22 and #23, there were tree iterators producing the results of a traversal. It was considerably more complex than a simple recursive traversal.
- Generators make it easier:

```
class BinTree:
    ...
    def preorder_values(self):
        if not self.is_empty:
            yield self.label
            yield from self.left.preorder_values()
            yield from self.right.preorder_values()
```

- The **yield from** G syntax takes a generator, G, and in effect performs:

```
for v in G: yield v
```

- It's really easy to change this to a postorder or inorder traversal!

Finite to Infinite

Currently, all our sequence data structures share common limitations:

- Each item must be explicitly represented, even if all can be generated by a common formula or function
- Sequence must be complete before we start iterating over it.
- Can't be infinite. Who cares?
  - "Infinite" in practical terms means "having an unknown bound".
  - Such things are everywhere.
  - Internet and cell phone traffic.
  - Instrument measurement feeds, real-time data.
  - Mathematical sequences.

Streams: A Lazy Structure

We'll define a *Stream* to look like an rlist whose `rest` is computed lazily.

```
class Stream(object):
    """A lazily computed recursive list."""

    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

    @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
empty_stream = Stream(None, None, True)
```

Example: The positive integers (all of them)

```
def make_integer_stream(first=1):
    """An infinite stream of increasing integers, starting at FIRST.
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)

>>> ints = make_integer_stream(1)
>>> ints.first
1
>>> ints.rest.first
2
```

Integer Streams in Action

- Initially, L=make_integer_stream(1) consists of one item with

```
L.first = 1, L._computed = False
```

- When we fetch L.rest, it becomes

```
L.first = 1, L._computed = True; L._rest = L2,
# where
L2.first = 2, L2._computed = False
```

- And so forth.

Mapping Streams

Familiar operations on other sequences can be extended to streams:

```
def map_stream(fn, s):
    """Stream of values of FN applied to the elements of stream S."""
    if s.empty:
        return s
    def compute_rest():
        return map_stream(fn, s.rest)
    return Stream(fn(s.first), compute_rest)

def combine_streams(fn, s0, s1):
    """Stream of the elements of S0 and S1 combined in pairs with
    two-argument function FN."""
    def compute_rest():
        return combine_streams(fn, s0.rest, s1.rest)
    if s0.empty or s1.empty:
        return empty_stream
    else:
        return Stream(fn(s0.first, s1.first), compute_rest)
```

Filtering Streams

Another example:

```
def filter_stream(fn, s):
    """Return a stream of the elements of S for which FN is true."""
    if s.empty:
        return s
    def compute_rest():
        return filter_stream(fn, s.rest)
    if fn(s.first):
        return Stream(s.first, compute_rest)
    return compute_rest()
```

A Few Conveniences

To look at streams a bit more conveniently, let's also define:

```
def truncate_stream(s, k):
    """A stream of the first K elements of stream S."""
    if s.empty or k == 0:
        return empty_stream
    def compute_rest():
        return truncate_stream(s.rest, k-1)
    return Stream(s.first, compute_rest)

def stream_to_list(s):
    """A list containing the elements of (finite) stream S."""
    r = []
    while not s.empty:
        r.append(s.first)
        s = s.rest
    return r
```

Finding Primes

```
def primes(pos_stream):
    """Return a stream of members of POS_STREAM that are not
    evenly divisible by any previous members of POS_STREAM.
    POS_STREAM is a stream of increasing positive integers.

    >>> p1 = primes(make_integer_stream(2))
    >>> stream_to_list(truncate_stream(p1, 7))
    [2, 3, 5, 7, 11, 13, 17]
    >>> p2 = primes(iterator_to_stream(positives()).rest)
    >>> stream_to_list(truncate_stream(p2, 7))
    [2, 3, 5, 7, 11, 13, 17]
    """
    def not_divisible(x):
        return x % pos_stream.first != 0
    def compute_rest():
        return primes(filter_stream(not_divisible, pos_stream.rest))
    return Stream(pos_stream.first, compute_rest)
```

Recursive Streams

What do you suppose we get from this?

```
f = Stream(1,
           lambda: Stream(1,
                          lambda: combine_streams(add, f, f.rest)))
stream_to_list(truncate_stream(f, 20))
```