Lecture 29: 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.

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.

Review: Iterables

- The Python for loop

for x in L:
  BODY

can use one of two strategies:

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ITER = L.iter()()</td>
<td>_L = 0, L</td>
</tr>
<tr>
<td>while True: try:</td>
<td>while True:</td>
</tr>
<tr>
<td>x = _ITER.next()()</td>
<td>x = _L[_I]</td>
</tr>
<tr>
<td>BODY</td>
<td>BODY</td>
</tr>
<tr>
<td>except StopIteration: break</td>
<td>_I += 1</td>
</tr>
<tr>
<td></td>
<td>except IndexError: break</td>
</tr>
</tbody>
</table>

- Crucial point: Iterators don’t compute items in a sequence until they are asked to. They are lazy (a technical term).

Iterables and Iterators

- Lists, dictionaries, and tuples are iterables; The __iter__ method on them yields an iterator.
- On standard iterators, the __iter__ method yields itself:

```python
>>> L = [1, 2, 3]
>>> it = L.__iter__()
>>> it is it.__iter__()
True
```
- This is useful, because several standard procedures (map, zip, re.finditer) return iterators, and several functions (like sum) use them:

```python
>>> L, P = [1, -2, 3], [2, 0, 4]
>>> map(abs, L)
<map object at 0x7f5f260dc2b0>
>>> sum(map(abs, L))
6
```
- map(abs, L)[0] ERROR (an iterator, not an iterable)

Generators: Another Kind of Iterator

- Generators provide a concise and elegant way to write iterators.
- Example: generator returning lists [0], [0, 1], [0, 1, 2], ...

```python
def triangle(n):
    """Generates all lists of the form [0], [0,1], ..., up to [0,...n-1]."""
    L = []
    for i in range(0, n):
        L += [i]
        yield L

>>> for p in triangle(3):
...     print(p)
... [0]
[0, 1]
[0, 1, 2]
```

Generators, explained

- A generator function is one that contains a yield statement.
- When called, a generator function returns a generator object.
- The generator object defines __next__, and acts like an iterator.
- When called, this __next__ function executes the body of the generator up to the next call to yield and then returns the result.
- On each subsequent call, starts from after the yield statement.
- Stops iterating on exit from the generator function.
Example With Trees

• Suppose we want to generate the labels of a tree in post-order (i.e., labels of children first, then label of node):

```python
def tree_labels(t):
    for c in t.children:
        for lab in tree_labels(c):
            yield lab
    yield t.label

• More succinctly, can write this as

```python
def tree_labels(t):
    for c in t.children:
        yield from tree_labels(c)
    yield t.label
```

• Now we can print all labels in a tree with

```python
for lab in tree_labels(t): print(lab)
```

Basic Stream Operations

```python
>>> s1 = Stream(1, lambda: Stream(2))
>>> s1.first
1
>>> s1.rest.first
2
>>> s1.rest.rest
Stream.empty
>>> def print_first(x): print("called"); return x
>>> s2 = Stream(1, lambda: print_first(Stream(2)))
>>> s2.rest.first
called
2
>>> s2.rest.first # .rest only computed first time called
2
```

Examples

An infinite stream of the same value.

```python
def make_const_stream(x):
    """An infinite stream of X's.""
    return Stream(x, lambda: make_const_stream(x))
```

The positive integers (all of them)

```python
def make_integer_stream(first=1):
    """The infinite stream FIRST, FIRST+1, ...""
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```

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

Mapping Streams

Familiar operations on other sequences can be extended to streams:

```python
def map_stream(fn, s):
    """Stream of values of FN applied to 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)
```

```python
def add_streams(s0, s1):
    """Stream of the sums of respective elements of S0 and S1."
    def add_streams(s0, s1):
        """Stream of the sums of respective elements of S0 and S1.""
        def compute_rest():
            return add_streams(s0.rest, s1.rest)
        if s0 is Stream.empty or s1 is Stream.empty:
            return Stream.empty
        else:
            return Stream(s0.first + s1.first, compute_rest)
```

Filtering Streams

Another example:

```python
def filter_stream(fn, s):
    """Return a stream of the elements of S for which FN is true.""
    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()
```
**Streams to Lists**

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

```python
def stream_to_list(s, n):
    """A list containing the elements of stream S,
    up to a maximum of N."""
    r = []
    while n > 0 and s is not Stream.empty:
        r.append(s.first)
        s = s.rest
        n -= 1
    return r
```

**Finding Primes**

```python
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(p1, 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 these?

```python
c1 = Stream(1, lambda: c1)
stream_to_list(c1, 5)
[1, 1, 1, 1, 1]

f1 = add_streams(c1, Stream(0, lambda: f1))
stream_to_list(f1, 5)
[1, 2, 3, 4, 5]

f2 = Stream(1,
            lambda: Stream(1,
                           lambda: add_streams(f2, f2.rest)))
stream_to_list(f2, 6)
[1, 1, 2, 3, 5, 8]
```