CS 61A Summer 2017

Coroutines

Discussion 12: July 31, 2017

1 Iterables & Iterators

An **iterable** is any container that can be processed sequentially. Examples include lists, tuples, strings, and dictionaries. Often we want to access the elements of an iterable, one at a time. We find ourselves writing `lst[0]`, `lst[1]`, `lst[2]`, and so on. It would be more convenient if there was an object that could do this for us, so that we don't have to keep track of the indices.

This is where **iterators** come in. We provide an iterable, using the `iter` function, and this returns a new iterator object. Each time we call `next` on the iterator object, it gives us one element at a time, just like we wanted. When it runs out of elements to give, calling `next` on the iterator object will raise a `StopIteration` error.

We can create as many iterators as we would like from a single iterable. But, each iterator goes through the elements of the iterable only once. If you want to go through an iterable twice, create two iterators!

```
>>> lst = [4, 2]
>>> i = iter(lst)
>>> j = iter(lst)
>>> i
<list_iterator object>
>>> next(i)
4
>>> next(i)
2
>>> next(j)
4
>>> next(i)
StopIteration
>>> next(j)
2
```

For Loops

By now, you are familiar with using `for` loops to iterate over iterables like lists and dictionaries.

This only works because the `for` loop implicitly creates an iterator using the built-in `iter` function. Python then calls `next` repeatedly on the iterator, until it raises `StopIteration`. The code to the right is (basically) equivalent to using a `for` loop to iterate over a list of `[4, 2]`.

```
>>> range_iterator = iter([4, 2])
>>> is_done = False
>>> while not is_done:
...     try:
...         val = next(range_iterator)
...         print(val)
...     except StopIteration:
...         is_done = True
4
2
```

Iterators as Classes

We can use object oriented programming to write a class that behaves like an iterator. There is an example implementation to the right.

To make a new instance of this `Iterator` class, you have to provide an iterable, just like you have to do with Python's built-in `iter` function.

Notice our `Iterator` class has a `__next__` method, so that we can call Python's built-in `next` on it to get the next element out of the iterable we initially passed in.

You might also notice there's an `__iter__` method. This may seem odd since we only use `iter` to obtain an iterator so why would we ever have to call `iter` on something that's already an iterator? Well, technically speaking, iterators are just a subcategory of iterables, since you are still able to iterate over them. Python wants every iterable including iterators themselves to support its built-in `iter` function. That's why we added an `__iter__` method that just returns `self`.

```
class Iterator:
    def __init__(self, lst):
        self.lst = lst
        self.i = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < len(self.lst):
            i += 1
            return self.lst[self.i]
        else:
            raise StopIteration
```

Questions

1.1 What does Python print?

```
>>> lst = [[1, 2]]
>>> i = iter(lst)
>>> j = iter(next(i))
>>> next(j)

>>> lst.append(3)
>>> next(i)

>>> next(j)

>>> next(i)
```

1.2 To make the `Link` class iterable, implement the `LinkIterator` class.

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest
    def __iter__(self):
        return LinkIterator(self)


class LinkIterator:
    def __init__(self, link):


    def __iter__(self):


    def __next__(self):
```

2 Generators

A **generator** function is a special kind of Python function that uses a **`yield`** statement instead of a **`return`** statement to report values. *When a generator function is called, it returns an iterator.* To the right, you can see a function that returns an iterator over the natural numbers. You can use **`yield from`** to take an iterator, and **`yield`** every value from that iterator.

When the `list` function in Python receives an iterator, it calls the `next` function on the input until it raises a `StopIteration`. It puts each of the elements from the calls to `next` into a new list and returns it.

```
>>> def gen_naturals():
...     current = 0
...     while True:
...         yield current
...         current += 1
>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

Questions

2.1 Write a generator function that returns all subsets of the positive integers from 1 to n. Each call to this generator's **next** method will return a list of subsets of the set [1, 2, ..., n], where n is the number of times **next** was previously called.

```
def generate_subsets():
    """
    >>> subsets = generate_subsets()
    >>> for _ in range(3):
    ...     print(next(subsets))
    ...
    [[]]
    [[], [1]]
    [[], [1], [2], [1, 2]]
    """
```

2.2 To make the Link class iterable, implement the __iter__ method using a generator.

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __iter__(self):
```

3 Streams

A **stream** is a lazily-evaluated linked list. A stream's elements (except for the first element) are only computed when those values are needed.

A Stream instance is similar to a Link instance. Both have first and rest attributes. The rest of a Link is either a Link or Link.empty. Likewise, the rest of a Stream is either a Stream or Stream.empty.

However, instead of specifying all of the elements in __init__, we provide a function, compute_rest, which will be called to compute the next element of the stream. Remember that the code in the function body is not evaluated until it is called, which lets us implement the desired evaluation behavior. It's very important that compute_rest should return a Stream, if you don't want your Stream to end.

```python
class Stream:
    empty = 'empty'

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        self.first = first
        self.cached_rest = None
        assert callable(compute_rest)
        self.compute_rest = compute_rest

    @property
    def rest(self):
        """Return the rest, computing it if necessary."""
        if self.compute_rest is not None:
            self.cached_rest = self.compute_rest()
            self.compute_rest = None
        return self.cached_rest

    def __repr__(self):
        rest = self.cached_rest if self.compute_rest is None else '<...>'
        return 'Stream({}, {})'.format(self.first, rest)
```

In the example below, we start out with a `Stream` whose first element is `n`, and whose `compute_rest` function creates another stream. When we do compute the `rest`, we get another `Stream`. It's first element will be `n+1`, and it's `compute_rest` function will create a third `Stream`. This third `Stream` will start at `n+2`, and its `compute_rest` will make a fourth `Stream`, and so on. We get an infinite stream of integers, computed one at a time.

```python
def naturals(n=0):
    return Stream(n, lambda: naturals(n+1))
```

Questions

3.1 Suppose you want an infinite stream of randomly generated numbers. Consider an attempt to implement this via the code below. Are there any problems with this? If so, how can we fix it?

```python
from random import random
random_stream = Stream(random(), lambda: random_stream)
```

3.2 Write a function `every_other`, which takes in an infinite stream and returns a stream containing its even indexed elements.

```python
def every_other(s):
```

3.3 Write a function `filter_stream`, which takes in a boolean function `f` and a `Stream` `s`. It should return a new `Stream`, containing only the elements of `s` for which `f` returns `True`.

```
def filter_stream(f, s):
```

3.4 Write a function `seventh` that creates an infinite stream of the decimal expansion of dividing n by 7. For example, the first 5 elements in `seventh(1)` would be 1, 4, 2, 8, and 5, since $1/7 = .14285$.

```
def seventh(n):
    """The decimal expansion of n divided by 7."""
```

4 Memoization in Streams

This implementation of streams also uses *memoization*, or caching. The first time a program asks a `Stream` for its `rest`, the `Stream` code computes the required value using `compute_rest`, saves the resulting value, and then returns it. After that, every time the `rest` is referenced, we simply return the value that we stored earlier.

Questions

4.1 What does Python print?

```
>>> foo = lambda x: 10 * x
>>> s = Stream(foo(1), lambda: foo(2))
>>> s

>>> foo = lambda x: 100 * x
>>> s.rest

>>> s

>>> s.compute_rest = lambda: Stream(foo(3), lambda: s)
>>> s

>>> s.rest

>>> s

>>> s.rest.rest
```

4.2 (Summer 2012 Final) What are the first five values in the following stream?

```
def my_stream():
    def compute_rest():
        return add_streams(filter_stream(lambda x: x % 2 == 0, my_stream()),
                           map_stream(lambda x: x + 2, my_stream()))
    return Stream(2, compute_rest)
```

5 Extra Practice

5.1 We can even represent the sequence of all prime numbers as an infinite stream! Define a function `sieve`, which takes in a stream of numbers and returns a new stream containing only those numbers which are not multiples of an earlier number in the stream. We can define `primes` by sifting all natural numbers starting at 2. Look online for the **Sieve of Eratosthenes** if you need some inspiration.

```
def sieve(s):
```

5.2 This is the function `combine_stream`. Use it to define the infinite stream of factorials below! You can assume `add` and `mul` have been imported, and you may also use the infinite stream of naturals from page 4.

```
def combine_stream(f, s1, s2):
    if s1 is Stream.empty or s2 is Stream.empty:
        return Stream.empty
    return Stream(f(s1.first, s2.first), lambda: combine_stream(f, s1.rest, s2.rest))

def evens():
    return combine_stream(add, naturals(0), naturals(0))

def factorials():
```

Now define a new `Stream`, where the nth term represents the degree-n polynomial expansion for e^x, which is $\sum_{i=0}^{n} x^i/i!$. You are allowed to use any of the other functions defined in this problem.

```
def exp(x):
```