CS61A Notes – Week 13 – Iterators, Generators, and Streams Iterators

An iterator is an abstract object that represents a sequence of values. An iterator understands how to do two things. First of all, it knows how to calculate its next value, and hence can continuously output a stream of its values. Secondly, an iterator should know when it has no more values left to compute. We can see these two behaviors being exhibited in Python's iterator interface. Python's iterator interface uses a `__next__` method that computes the next value in the sequence given the iterator's current state. Calling `next(<some-iterator>)` will implicitly call the `__next__` method, causing the iterator to return its next value. When there are no more values left to compute, the `__next__` method is required to raise a `StopIteration` Exception, thus signaling the end of the value sequence.

The `__iter__` method is another method used in Python's iterator interface, and it merely returns the iterator object itself. In other words, calling `__iter__` returns an object that implements the `__next__` method.

Keep in mind that iterators are stateful objects and always keep track of how much of the stream has already been computed. Once an iterator invokes its `__next__` method, the calculated value gets "used-up" and thrown away, hence changing the iterator's state. If we then make another call to __next__, the returned value will be different from the previously calculated value.

Here is an example of a class that implements Python's iterator interface. This iterator calculates all of the natural numbers one-by-one, starting from zero:

```
class Naturals():
    def __init__(self):
        self.current = 0

    def __next__(self):
        result = self.current
        self.current += 1
        return result

    def __iter__(self):
        return self
```

Notice that the __next__ method in Naturals never raises a StopIteration exception. Wait, but I thought iterators have to raise a StopIteration exception when they compute all of their values! Exactly, but it doesn't make sense to "compute the last natural number", as the set of natural numbers is infinitely large. Hence, we see here that iterators are perfectly capable of representing an infinite data stream using a finite amount of memory. Also note that because the return statement must be the last one executed in the __next__ method, we have to save the return value in a temporary variable, or else it would get overwritten by the intervening update step. Later on we will introduce python's yield statement, which lets you define iterators more naturally.

Questions

1. Define an iterator whose i-th element is the result of combining the i-th elements of two input streams using some binary operator, also given as input. The resulting iterator should have a size equal to the size of the shorter of its two input iterators.

```
>>> from operator import add
>>> evens = Iter_Combiner(Naturals(), Naturals(), add)
>>> next(evens)
0
>>> next(evens)
2
>>> next(evens)
4

class Iter_Combiner():
    def __init__(self, iter1, iter2, combiner):



    def __next__(self):




    def __iter__(self):
        return self
```

2. What results from executing this sequence of commands?

```
>>> naturals = Naturals()
>>> doubled_naturals = Iter_Combiner(naturals, naturals, add)
>>> next(doubled_naturals)
_________
>>> next(doubled_naturals)
_________
```

3. Create an iterator that generates the sequence of Fibonacci numbers.

```
class Fibonacci_Numbers():
    def __init__(self):



    def __next__(self):




    def __iter__(self):
        return self
```

Generators

A generator is a special kind of Python iterator that uses a yield statement instead of a return statement to report values. A yield statement is similar to a return statement, but whereas a return statement causes the current environment to be destroyed after a function exits, a yield statement inside a function causes the environment to be saved until the next time `__next__` is called, which allows the generator to automatically keep track of the iteration state. Once `__next__` is called again, execution picks up from where the previously executed yield statement left off, and continues until the next yield statement (or the end of the function) is encountered. Including a yield statement in a function automatically signals to Python that this function will create a generator. In other words, when we call the function, it will return a generator object, instead of executing the code inside the body. When the returned generator's `__next__` method is called, the code in the body is executed for the first time, and stops executing upon reaching the first yield statement.

A python function can either use return statements or yield statements in the body to output values, but not both. If the function has a single yield statement, then python treats the function as a generator, and will raise an error if the body also contains a return expression that outputs a value.

Here is an iterator for the natural numbers written using the generator construct:

```
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
```

Questions

1. In this problem we will represent a set using a python list. Write a generator function that returns lists of 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.

```
>>> subsets = generate_subsets()
>>> next(subsets)
[[]]
>>> next(subsets)
[[], [1]]
>>> next(subsets)
[[], [1], [2], [1, 2]]

def generate_subsets():
```

2. Define a generator that yields the sequence of perfect squares.

```
def perfect_squares():
```

3. Remember the hailstone sequence from homework 1? Implement it using a generator! To generate a hailstone sequence:

Pick a positive number n
If n is even, divide it by 2
If n is odd, multiply it by 3 and add 1
Continue this process until n is 1

```
def generate_hailstone(n=10):
```

Streams

A stream is our third example of a lazy sequence. **A stream is a lazily evaluated rlist.** In other words, the stream's elements (except for the first element) are only evaluated when the values are needed. Hence, as with iterators and generators, streams can be used to represent an infinite amount of data without using an infinite amount of memory.

Take a look at the following code from lecture:

```
class Stream(object):
    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)
```

We represent Streams using Python objects, similar to the way we defined rlists. We nest streams inside one another, and compute one element of the sequence at a time. Note that instead of specifying all of the elements in __init__, we provide a function, compute_rest, that encapsulates the algorithm used to calculate the remaining elements 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. This implementation of streams also uses memoization. The first time a program asks a stream for its `rest` field, the stream code computes the required value using `compute_rest`, saves the resulting value, and then returns it. After that, every time the rest field is referenced, the stored value is simply returned and it is not computed again.

Here is an Example:

```
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```

Notice what is happening here. We start out with a stream whose first element is 1, and whose compute_rest function creates another stream. So when we do compute the rest, we get another stream whose first element is one greater than the previous element, and whose compute_rest creates another stream. Hence we effectively get an infinite stream of integers, computed one at a time. This is almost like an infinite recursion, but one which can be viewed one step at a time, and so does not crash.

Questions:

1. What modification to the Stream class could be made to allow us to define an infinite stream of random numbers?

3. Write a procedure make_fib_stream() that creates an infinite stream of Fibonacci Numbers. Make the first two elements of the stream 0 and 1.
Hint: Consider using a helper procedure that can take two arguments, then think about how to start calling that procedure.

```
def make_fib_stream()
```

2. Write a procedure sub_streams that takes in two streams as argument, and returns the Stream of corresponding elements from the second stream subtracted from the first stream. You can assume that both Streams are infinite.

```
def sub_streams(stream1, stream2):
```

4. Define a procedure that inputs an infinite Stream, s, and a target value and returns True if the stream converges to the target within a certain number of values. For this example we will say the stream converges if the difference between two consecutive values and the difference between the value and the target drop below max_diff for 10 consecutive values. (Hint: create the stream of differences between consecutive elements using sub_streams)

```
def converges_to(s, target, max_diff=0.00001, num_values=100):
```

Higher Order Functions on Streams:
Naturally, as the theme has always been in this class, we can abstract our stream procedures to be higher order. Take a look at filter_stream from lecture:

```
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if stream.empty:
        return stream
    elif filter_func(stream.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_funct, stream.rest)
```

You can see how this function might be useful. Notice how the Stream we create has as its `compute_rest` function a procedure that "promises" to filter out the rest of the Stream when asked. So at any one point, the entire stream has not been filtered. Instead, only the part of the stream that has been referenced has been filtered, but the rest will be filtered when asked. We can model other higher order Stream procedures after this one, and we can combine our higher order Stream procedures to do incredible things!

Questions:

1. In a similar model to filter_stream, write a procedure map_stream, that takes in a one-argument function, and does the obvious thing.

```
def stream_map(func, stream):
```

2. The following procedure when called creates a Stream. Given endless subsequent calls to the Stream to compute all its values, what are the values of the Stream created by the call to my_stream(). Don't write them all, just write enough to get the pattern.

```
def my_stream():
    def compute_rest():
        return add_streams(map_stream(double, my_stream()), my_stream())
    return Stream(1, compute_rest)
```