1 Introduction

Infinite sequences frequently arise in computer science. For example, the sequence of all natural numbers is an infinite sequence, because there is no “last” natural number. However, it is impossible to physically store an infinite amount of data. How do we get around this?

In this section, we will learn about iterators, generators, and streams – each of these constructs is designed to represent infinite sequences in a finite amount of memory.

2 Iterators

An iterator is an object that represents a sequence of values. 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:

```python
class Naturals():
    def __init__(self):
        self.current = 0
    def __next__(self):
        result = self.current
        self.current += 1
        return result
    def __iter__(self):
        return self
```
There are two components of Python’s iterator interface: the \_next\_ method, and the \_iter\_ method.

### 2.1 \_next\_

The \_next\_ method does two things:

1. calculates the next value
2. checks if it has any values left to compute

To return the next value in the sequence, the iterator does some computation defined in the \_next\_ method.

When there are no more values left to compute, the \_next\_ method must raise a type of exception called StopIteration. This signals the end of the sequence.

**Note:** the \_next\_ method defined above does NOT raise any StopIteration exceptions. Why? Because there are always more values left to compute! Remember, there is no “last natural number”, so there is technically no “end of the sequence.” However, if you wanted to define a finite iterator, then you would raise a StopIteration after returning the final value.

### 2.2 \_iter\_

The purpose of the \_iter\_ method is to return an iterator object. **By definition,** an iterator object is an object that has implemented both the \_next\_ and \_iter\_ methods.

This has an interesting consequence. If a class implements both a \_next\_ method and a \_iter\_ method, its \_iter\_ method can just return self (like in the example). Since the class implements both \_next\_ and \_iter\_, it is technically an iterator object, so its \_iter\_ method can just return itself.

### 2.3 Implementation

When defining an iterator object, you should always keep track of how much of the sequence has already been computed. In the above example, we use an instance variable self.current to keep track.

Iterator objects maintain state. Successive calls to \_next\_ will most likely output different values each time, so \_next\_ is considered non-pure.

How do we call \_next\_ and \_iter\_? Python has built-in functions called next and iter for this. Calling next(some_iterator) will then cause Python to implicitly call some_iterator’s \_next\_ method. Calling iter(some_iterator) will make a similar implicit call to some_iterator’s \_iter\_ method.
For example, this is how we would use the `Naturals` iterator:

```python
>>> nats = Naturals()
>>> nats_iter = iter(nats)
>>> next(nats_iter)
0
>>> next(nats_iter)
1
>>> next(nats_iter)
2
```

However, we don’t really need to call `iter` on `nats`. Why not?

One other note: you can use iterator objects in `for` loops. In other words, any object that satisfies the iterator interface can be iterated over:

```python
>>> nats = Naturals()
>>> for n in nats:
    print(n)
0
1
2
...  # Forever!
```

This works because the Python `for` loop implicitly calls the `__iter__` method of the object being iterated over, and repeatedly calls `next` on it. In other words, the above interaction is (basically) equivalent to:

```python
nats_iter = iter(nats)
is_done = False
while not is_done:
    try:
        val = next(nats_iter)
        print(val)
    except StopIteration:
        is_done = True
```

### 2.4 Questions

1. Define an iterator whose $i$-th element is the result of combining the $i$-th elements of two input iterables 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.

```python
>>> 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):
        self.iter1 = iter(iter1)
        self.iter2 = iter(iter2)
        self.combiner = combiner

    def __next__(self):
        return self.combiner(next(self.iter1), next(self.iter2))

    def __iter__(self):
        return self

2. What is the result of executing this sequence of commands?

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

Solution: 1

>>> next(doubled_naturals)
3. Create an iterator that generates the sequence of Fibonacci numbers.

```python
class Fibonacci_Numbers():
    def __init__(self):
        self.current = 0
        self.next = 1

    def __next__(self):
        res = self.current
        self.current, self.next = self.next, self.current + self.next
        return res

    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.

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

```python
def generate_naturals():
    current = 0
    while True:
        yield current
        current += 1
```
Calling `generate_naturals()` will return a generator object:

```python
>>> gen = generate_naturals()
>>> gen
<generator object gen at ...>
```

To use the generator object, you then call `next` on it:

```python
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
```

Think of a generator object as containing an implicit `_next_` method. This means, by definition, a generator object is an iterator.

### 3.1 yield

The `yield` statement is similar to a `return` statement. However, while a `return` statement causes the current environment to be destroyed after a function exits, a `yield` statement 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. 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. **Having both will raise an error.**

### 3.2 Implementation

Because generators are technically iterators, you can implement `_iter_` methods using only generators. For example,

```python
class Naturals:
    def __init__(self):
```
self.current = 0
def __iter__(self):
    while True:
        yield self.current
        self.current += 1

The usage of a Naturals object is exactly the same as before:
>>> nats = Naturals()
>>> nats_iter = iter(nats)
>>> next(nats_iter)
0
>>> next(nats_iter)
1
>>> next(nats_iter)
2

There are a couple of things to note:

• No __next__ method in Naturals: Remember, __iter__ just has to return an object that has implemented a __next__ method. Since generators have their own __next__ method, the new Naturals implementation is perfectly valid.

• nats is a Naturals object – nats_iter is a generator: do not treat nats as the iterator!

Since generators are iterators, you can also use generators in for loops.

3.3 Questions

1. 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 \([1, 2, \ldots, n]\), where \(n\) is the number of times __next__ was previously called.

>>> subsets = generate_subsets()
>>> next(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.

```python
def perfect_squares():
    i = 0
    while True:
        yield i * i
        i += 1
```

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

```python
def generate_hailstone(n=10):
    while n != 1:
        yield n
        if (n % 2) == 0:
            n = n / 2
        else:
            n = 3 * n + 1
        yield n
```
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.

Take a look at the following code:

```python
class Stream(object):
    class empty(object):
        def __repr__(self):
            return 'Stream.empty'

    empty = empty()

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

    def __repr__(self):
        return 'Stream([0], <...>).format(repr(self.first))
```

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:

```python
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.

### 4.1 Questions

1. 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.

   ```python
def make_fib_stream():
    Solution:
    return fib_stream_generator(0, 1)

def fib_stream_generator(a, b):
    def compute_rest():
        return fib_stream_generator(b, a+b)
    return Stream(a, compute_rest)
```

2. Write a procedure `sub_streams` that takes in two streams `s1`, `s2`, and returns a new stream that is the result of subtracting elements from `s1` by elements from `s2`. For instance, if `s1` was `(1, 2, 3, ...)` and `s2` was `(2, 4, 6, ...),` then the output would be the stream `(-1, -2, -3, ...).` You can assume that both Streams are infinite.

   ```python
def sub_streams(s1, s2):
```
3. 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)

```python
def converges_to(s, target, max_diff=0.00001, num_values=100):
    count = 0
    deriv = sub_streams(s.rest, s)
    for i in range(num_values):
        if abs(s.first - target) <= max_diff and abs(deriv.first) <= max_diff:
            count += 1
        else:
            count = 0
        if count == 10:
            return True
        deriv = deriv.rest
        s = s.rest
    return False
```

4.2 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:

```python
def filter_stream(filter_func, stream):
    def make_filtered_rest():
        return filter_stream(filter_func, stream.rest)
    if Stream.empty:
        return Stream(None, None)
    def rest():
        return make_filtered_rest().first
    def first():
        return filter_func(stream.first)
    return Stream.first(rest, first)
```


```python
return stream
elif filter_func(stream.first):
    return Stream(stream.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!

### 4.3 Questions

1. In a similar model to `filter_stream`, let’s recreate the procedure `map_stream` from lecture, that given a stream `stream` and a one-argument function `func`, returns a new stream that is the result of applying `func` on every element in `stream`.

   ```python
def stream_map(func, stream):
    
    def compute_rest():
        return stream_map(func, stream.rest)
    if stream.empty:
        return stream
    else:
        return Stream(func(stream.first), compute_rest)
    ```

   Solution:
2. What does the following Stream output? Try writing out the first few values of the stream to see the pattern.

```python
def my_stream():
    def compute_rest():
        return addStreams(mapStream(double,
                                  my_stream()),
                       my_stream())

    return Stream(1, compute_rest)
```

Solution: Powers of 3: 1, 3, 9, 27, 81, ...