Solutions to CS 61A Challenge Problems: Iterators, Iterables, and Generators (and Streams)

Other worksheets and solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

1 Iterators and Iterables

(a) Complete the following class `PrimeIterator` so that it correctly iterates through the prime numbers in the interval `[start, end)` one by one. You may assume that the function `is_prime` is already written for you.

```
class PrimeIterator:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.current = start

    def __next__(self):
        if self.current < self.end:
            while not is_prime(self.current):
                self.current += 1
            if self.current < self.end:
                self.current += 1
                return self.current - 1
        raise StopIteration()
```

The `__next__` method simply returns the subsequent values needed. We store our current progress as an instance variable. Don't forget to raise a `StopIteration` exception when we want to stop.

(b) If we wanted to use a `PrimeIterator` instance in a `for` loop or in the function `list`, then what method do we need to add? Write this method.
`for` loops and functions like `list` and `zip` operate on iterables, not iterators. A class is iterable if it has an `__iter__` method that returns an iterator. Since `PrimeIterator` is already an iterator, it can just return itself.

```
def __iter__(self):
    return self
```

In practice, it is common to have a class that is both an iterator and an iterable. This keeps things simple, since all iteration methods will now work on it.

(c) `p = PrimeIterator(2, 15)`. Eventually what happens if we keep calling `next` on `p`?
Eventually, we'll reach the `StopIteration` exception and Python will throw us this exception.

(d) Complete the `PrimeIterable` class that is an iterable and has the same functionality as `PrimeIterator`. This means that we can use it in a `for` loop, call `list` on it, etc.

```
class PrimeIterable:
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.current = start

    def __getitem__(self, i):
        # Assume this is already implemented for you
    def __iter__(self):
        return PrimeIterator(self.start, self.end)
```

An iterable simply needs an `__iter__` method that returns an iterator. In this case, all we need to do is return the corresponding iterator. As an aside, it is not often clear whether or not an iterable needs to have an `__iter__` method, a `__getitem__` method, or both. Older versions of Python define iterables as something you can index into, but newer versions prefer the `__iter__` method. It is oftentimes safest to simply implement both.

(e) `q = PrimeIterable(2, 15)`. What happens if keep calling `next` on `q`? What is the result of calling `list(q)`? There is no `next` functionality for `q`, so we will get a `TypeError` saying that `PrimeIterable` is not an iterator. But since it is an iterable, we can definitely call `list` on it, which gives us `[2, 3, 5, 7, 11, 13]`.

(f) Implement the class `Vowels` that takes in a word and allows you to step through each of the vowels in the word in order. `Vowels` *is both an iterator and an iterable* that also supports indexing.

```
import re
def get_vowels(word):
    # A bit of RegEx magic to isolate all the vowels of a word in order
    return re.sub(r'[^aeiou]', '', word)

class Vowels:
    def __init__(self, word):
        self.vowels = get_vowels(word)
        self.counter = 0

    def __next__(self):
        if self.counter < len(self.vowels):
            self.counter += 1
            return self.vowels[self.counter-1]
        raise StopIteration

    def __iter__(self):
        return self

    def __getitem__(self, i):
        try:
            return self.vowels[i]
        except IndexError:
            raise IndexError("Out of bounds index " + str(i))
        except TypeError as e:
            raise TypeError(e)
```

Again, we need to store our current progress for the `next` method, and we raise a `StopIteration` exception at the end of the string. Here we have both `__iter__` and `__getitem__`. Notice that in the latter, we handle exceptions differently if the index given is out of bounds or not an integer. The `IndexError` gets a special message passed in, whereas the `TypeError` is simply constructed from the original error object.

(g) What is the result of the following code?

```
list(Vowels("facetious"))
```

['a', 'e', 'i', 'o', 'u']

```
Vowels("aardvark")[3]
IndexError:  Out of bounds index 3
```

```
next(Vowels("sciatic"))
```
'i'

2 Generators

(a) Write the generator function `randoms` that can generate `num` random integers in the interval `[low, high)`.

```
from random import random

def randoms(low, high, num):
    for _ in range(num):
        yield int((random() * (high - low)) + low)
```

The `yield` statement is a bit tricky. When we hit a `yield`, it will spit out the value, and it will save the place of execution. When we call `next` on this generator again, execution will continue after the last `yield` left off, and continue until the next one. At the very end of the generator, when there is no more code to run, a `StopIteration` exception is automatically thrown.

(b) Write a generator expression that gives the same result as the function.
`(int((random() * (high - low)) + low) for _ in range(num))`
This is not a tuple comprehension. It is a generator expression. Similar to setting `r = randoms(1, 10, 5)`, this expression also yields a generator that has the exact same functionality as `randoms`. Note that generators are iterators themselves, and they are also iterable, since they can be passed into `list` and used in `for` loops, etc.

3 Streams

What is the 4th element in this stream? Assume 1-indexing.

```
(define (mystery foo)
  (let ((bar (+ (* foo 3) 1)))
    (cons-stream bar
                 (mystery bar))))

(mystery 3)
```

This is an infinite stream that starts with a certain value `foo`. It is important to write the recurrence relation. If $M[n]$ is the nth item in the stream, then $M[n] = 3M[n-1]+1$. Since we are using 1-indexing, $M[1] = 3$. Then we simply walk through the stream until the 4th element:
$M[1] = 3$
$M[2] = 3(3) + 1 = 10$
$M[3] = 3(10) + 1 = 31$
$M[4] = 3(31) + 1 = 94$