Review of Iterators

Recall that something is *iterable* if it supports the `iter` function—that is the method `__iter__` is defined—and returns an iterator. An *iterator* is something that

- supports the `next` function—that is, the method `__next__` is defined;
- throws a `StopIteration` when the iterator is empty; and
- returns itself under an `iter` call.

Iterators may be defined using *classes* or with *generators*. For example, suppose we want an iterator that generates all squares below a certain threshold. We could define the following `squares` class.

```
class squares:

    def __init__(self, threshold=None):
        self._state = 1
        self._threshold = threshold

    def _below_threshold(self):
        return self._threshold is None or self._state**2 < self._threshold

    def __iter__(self):
        return self

    def __next__(self):
        if self._below_threshold():
            sq = self._state**2
            self._state += 1
            return sq
        else:
            raise StopIteration()
```

Some specific points:

- We use the optional parameter `threshold=None` to allow for infinite generation. This convention of setting the value to `None` is common in Python.
- The `__iter__` method returns `self`
- The `_below_threshold` method makes use of a *short-circuited* logical `or` operator. Short-circuited means that the expressions are evaluated left-to-right and if the whole expression can be inferred without evaluating any more expressions, then evaluation is complete. In this case, if `self._threshold is None` then the right-hand side is never evaluated, which is good because you can't compare an integer to `None`.
- The `__next__` method raises a `StopIteration` using the `raise` syntax.

We could also define the following generator.

```
def squares_gen(threshold=None):
  i = 1
  while threshold is None or i**2 < threshold:
    yield i**2
    i += 1
```

Class Exercise: Powers of k

Define an iterator for powers of k with an optional second argument `length` argument specifying how many of the first k powers to generate)

```
def powers_of(k, length=None):
  i = 0
  while length is None or i < length:
    yield k**i
    i += 1
```

Inheritance and Overriding Methods

Without getting too technical, the primary characteristics associated with object-oriented programming are

- inheritance;
- encapsulation; and
- polymorphism

Inheritance is a mechanism by which a class retains the state and behavior of another class. Encapsulation is about creating a public interface for your class and keeping the internal state sequestered. Polymorphism just means that a class is free to override a method from its base class and that the correct version of the method always gets called. In python, there is direct support for inheritance, encapsulation happens via naming conventions, and polymorphism happens by default—the most specific version of a method is always called, but one can use `super()` to refer to the super class.

Example: Even Squares

Imagine that you wanted to create an iterator that returned squares that were even. One way to do this is to create a new `even_squares` class that *inherits* from `squares`. Without any new methods, the `even_squares` class inherits the behavior of `squares` as is. However, when `next` is called, we only want even squares returned. To do this, we *override* the the `__next__` method so that it calls the next method of its *superclass* until it reaches an even square.

```
class even_squares(squares)

  def __next__(self):
  sq = super().__next__()
  while (sq % 2 != 0):
   sq = next(super())
  return sq
```