Lecture 25: Iterators and Generators
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 an object 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*. 
class SquaresIter:

    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()
class EvenSquaresIter(SquaresIter):

    def __next__(self):
        sq = super().__next__()
        while (sq % 2 != 0):
            sq = super().__next__()
        return sq
It is possible (and common) to exhaust an iterator’s data:

```python
>>> si = SquaresIter(10)
>>> si
<SquaresIter object at 0x7f2ae6fd9278>
>>> list(si)
[1, 4, 9]
>>> list(si)
[]
```

By nature, `__next__()` moves an object’s internal state in one direction: forward.
We may want to define iterable classes that are not iterators themselves.

```python
class Squares:
    def __init__(self, threshold=None):
        self._threshold = threshold

    def __iter__(self):
        return SquaresIter(self._threshold)
```

```python
>>> sq = Squares(10)
>>> sq
<Squares object at 0x7fb529e3c2b0>
>>> list(sq)
[1, 4, 9]
>>> list(sq)
[1, 4, 9]
```
We may want to define iterable classes that are not iterators themselves.

```python
class Squares:
    def __init__(self, threshold=None):
        self._threshold = threshold

    def __iter__(self):
        return SquaresIter(self._threshold)

>>> sq = Squares(10)
>>> sq
<Squares object at 0x7fb529e3c2b0>
>>> list(sq)
[1, 4, 9]
>>> list(sq)
[1, 4, 9]
```
We have modified our functions to print each time they are executed in order to see what is happening internally:

```python
>>> sq = Squares(10)
Squares: __init__()
>>> list(si)
Squares: __iter__()
SquaresIter: __init__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: raise StopIteration()
[1, 4, 9]
```
We have modified our functions to print each time they are executed in order to see what is happening internally:

```python
>>> sq = Squares(10)
Squares: __init__()
>>> list(si)
Squares: __iter__()
SquaresIter: __init__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: __next__()
SquaresIter: raise StopIteration()
[1, 4, 9]
```

An individual iterator may exhaust its data, but the Squares object just create a new one when `iter()` is called.
Instead of the `return` keyword, generators use `yield`

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

A `yield` statement passes control back to the calling function, but it *preserves the local state of the function*.
A generator function returns an object that behaves just like an iterator.

```python
>>> sg = squares_gen(10)
>>> sg
<generator object squares_gen at 0x7f16396dbd58>
>>> next(sg)
1
>>> next(sg)
4
>>> next(sg)
9
>>> next(sg)
Traceback (most recent call last):
  File "<stdin>" , line 1, in <module>
StopIteration
>>> 
>>> sg = squares_gen(10)
>>> sg
>>> list(sg)
[1, 4, 9]
>>> list(sg)
[]
```
Define an iterator for powers of $k$ with an optional second argument $\text{length}$ argument specifying how many of the first $k$ powers to generate.
Define an iterator for powers of \( k \) with an optional second argument `length` argument specifying how many of the first \( k \) powers to generate.

```python
class PowersOfK:
    def __init__(self, k, length=None):
        self._k = k
        self._pow = 0
        self._length = length

    def _below_threshold(self):
        return self._length is None or self._pow < self._length

    def __iter__(self):
        return self

    def __next__(self):
        if self._below_threshold():
            v = self._k**self._pow
            self._pow += 1
            return v
        else:
            raise StopIteration()
```
Define a generator function for powers of $k$ with an optional second argument `length` argument specifying how many of the first $k$ powers to generate.
Define a generator function for powers of $k$ with an optional second argument `length` argument specifying how many of the first $k$ powers to generate.

```python
def powers_of_k(k, length=None):
    """
    generator for powers of $k$
    Args:
        $k$ (int): base that we exponentiate
        length (int): how many of the first $k$ powers to generate
    """
    i = 0
    while length is None or i < length:
        yield k**i
        i += 1
```