Lecture 25: Iterators and Generators

# Iterators - Review

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*.

# An Iterator for Squares

```
1  class SquaresIter:
2
3      def __init__(self, threshold=None):
4          self._state = 1
5          self._threshold = threshold
6
7      def _below_threshold(self):
8          return self._threshold is None or self._state**2 < self._threshold
9
10     def __iter__(self):
11         return self
12
13     def __next__(self):
14         if self._below_threshold():
15             sq = self._state**2
16             self._state += 1
17             return sq
18         else:
19             raise StopIteration()
```

# An Iterator for Even Squares

```
1  class EvenSquaresIter(SquaresIter):
2
3      def __next__(self):
4          sq = super().__next__()
5          while (sq % 2 != 0):
6              sq = super().__next__()
7          return sq
```

# Separating Iterables from Iterators

It is possible (and common) to exhaust an iterator's data:

```
>>> 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.

# Separating Iterables from Iterators

We may want to define iterable classes that are not iterators themselves.

```
1  class Squares:
2      def __init__(self, threshold=None):
3          self._threshold = threshold
4
5      def __iter__(self):
6          return SquaresIter(self._threshold)
```

# Separating Iterables from Iterators

We may want to define iterable classes that are not iterators themselves.

```
1  class Squares:
2      def __init__(self, threshold=None):
3          self._threshold = threshold
4
5      def __iter__(self):
6          return SquaresIter(self._threshold)
```

```
>>> sq = Squares(10)
>>> sq
<Squares object at 0x7fb529e3c2b0>
>>> list(sq)
[1, 4, 9]
>>> list(sq)
[1, 4, 9]
```

# Separating Iterables from Iterators

We have modified our functions to print each time they are executed in order to see what is happening internally:

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

# Separating Iterables from Iterators

We have modified our functions to print each time they are executed in order to see what is happening internally:

```
>>> sq = Squares(10)
Squares: __init__()
>>> list(si)
Squares: __iter__()
SquaresIter: __init__()
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.

# A Generator for Squares

Instead of the `return` keyword, generators use `yield`

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

A `yield` statements passes control back to the calling function, but it *preserves the local state of the function*

# A Generator for Squares

A generator function returns an object that behaves just like an iterator.

```
>>> 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)
[]
```

# 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.

# 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.

```
1  class PowersOfK:
2
3      def __init__(self, k, length=None):
4          self._k = k
5          self._pow = 0
6          self._length = length
7
8      def _below_threshold(self):
9          return self._length is None or self._pow < self._length
10
11     def __iter__(self):
12         return self
13
14     def __next__(self):
15         if self._below_threshold():
16             v = self._k**self._pow
17             self._pow += 1
18             return v
19         else:
20             raise StopIteration()
```

# Class Exercise: Powers of $k$

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.

# Class Exercise: Powers of $k$

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.

```
1  def powers_of_k(k, length=None):
2      """
3      generator for powers of k
4      Args:
5          k (int): base that we exponentiate
6          length (int): how many of the first k powers to generate
7      """
8      i = 0
9      while length is None or i < length:
10         yield k**i
11         i += 1
```