Iterators and Generators

More on Iterators

A Python object is *iterable* if it supports the `iter` function—that is, it has the magic method `__iter__` defined—and returns an iterator object. An *iterator* is an object that supports the `next` function—that is, it has the magic method `__iter__` defined—, throws a `StopIteration` when the iterator is empty, and returns itself under an `iter` call.

Last lecture we defined an iterator class that yields squares of consecutive integers that fall below some threshold. Here it is again for reference.

```
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()
```

Suppose we want to define another iterator class that yields squares of consecutive integers that fall below some threshold *and are even*. We can take advantage of inheritance to reuse much of this code. Here is an iterator class `EvenSquaresIter` that inherits from the `SquaresIter` class.

```
class EvenSquaresIter(SquaresIter):

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

- The `EvenSquaresIter` class inherits the behavior of `SquaresIter` for all methods by default, including the `__init__` method.
- We *override* the the `__next__` method so that it calls the next method of its *superclass* until it reaches an even square.
- Since the `SquaresIter` class's `__next__` method raises a `StopIteration` exception, that exception is passed to our `EvenSquaresIter` class's `__next__` method, which is in turn passed to its caller.

So far we have defined iterable classes that were themselves iterators. But consider what happens when we try to use the `SquaresIter` iterator to create multiple lists.

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

This is becuase our `SquaresIter` class, like many iterators, yields items in order until all items exhausted, at which point it raises a StopIteration exception. By its nature, the `__next__()` method moves an object's internal state in one direction: forward.

We can, however, separate our iterable class from its iterator. In this way, we can create iterables whose data is not exhausted.

```
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. This lets us see what is happening internally. It looks something like this:

```
>>> 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]
```

Creating a `list` first calls `iter()` to create an iterator object. Then it calls `next()` on that iterator object until a `StopIteration` exception is raised. So each time we create a list from a SquaresIter object, it returns a new iterator. An individual iterator may exhaust its data, but the `Squares` object just create a new one when `iter()` is called.

Generators

We can accomplish the same tasks using a *generator*. Let's reexamine our previous example where we want to generate all squares below a certain threshold. We could use the following generator function:

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

Normally, a function exectes line by line, in order, until a `return` statement is found (or the function ends). `return` causes the flow of our code to leave the function; the state of all local variables is lost, and a single value is passed to the fucntion's caller.

The `yield` statement is similar to return in that it causes the flow of code to leave the function, and passes a single value to the caller. However, the function state is preserved. We may renter the function at the exact same point, and all of the local variables' values are restored.

Generators are a powerful tool that let us create iterable objects with very few lines of code.

```
>>> 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.

```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.

```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
```