iterator/generator

- what really is enumerate(a)? not a list!
- what is the difference b/w range(10) and xrange(10)?
- an iterator is a lazy list
- generator is the easiest way to implement an iterator

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

how to traverse a BST?

- lazy version of BST.list() (useful in HW2)

```
    def iteritems(self):
        if self.val is None:
            return
        for item in self.left.iteritems():
            yield item
        yield self.val
        for item in self.right.iteritems():
            yield item
```

generator expression

- lazy version of list comprehension
- memory efficient, but does not support list operations

```
# Generator expression (lazy)
(x*2 for x in xrange(256))

# List comprehension (non-lazy)
[x*2 for x in range(256)]
```

```
print enumerate(range(10))[:2]     # can’t index or slice
print [5,6] + enumerate(range(10)) # can't be added to lists
```

lazy versions

- range vs. xrange
- zip vs. itertools.izip
- dict.items() vs. dict.iteritems()
- dict.keys() vs. dict
- file.readlines() vs. file

```
In [3]: %timeit range(100000)
1000 loops, best of 3: 956 us per loop

In [4]: %timeit xrange(100000)
10000000 loops, best of 3: 180 ns per loop

In [1]: %timeit range(1000)
100000 loops, best of 3: 8.22 us per loop

In [2]: %timeit xrange(1000)
10000000 loops, best of 3: 179 ns per loop

Note: xrange(...) takes constant time!
```

iterators: the next() method

- lazy lists are actually lazy linked lists, with a next method
- xrange() returns a lazy list, but not an iterator
  - designed for backward compatibility; fixed in Python 3
  - iter(xrange(...)) returns an iterator
- two ways to iterate through an iterator
  - for ... in ...: ... # no need for explicit use of next()
  - while True:
    - try: ... a.next() ...
    - except: break

implementing izip

```
def izip2(a, b):  # a, b are lists or lazy lists
    a = iter(a)
    b = iter(b)
    while True:
        try:
            aa = a.next()
            bb = b.next()
            yield (aa, bb)
        except:
            break

>>> a = izip2(range(3), xrange(5))
>>> while True:
...    print a.next()
...
(0, 0)
(1, 1)
(2, 2)
StopIteration Error  # by izip2, not by iter(range(3)).next()!
```

Summary: 3 ways to make an iterator

- 1. generator expression: `(x*2 for x in xrange(256))`
- 2. yield statement
- 3. implementing next() and __iter__() methods

```
>>> class A(object):
...     def __init__(self, a):
...         self.A = range(a)
...         self.idx = -1
...
...     def __iter__(self): # make an A instance iterable
...         return self
...
...     def next(self):
...         self.idx += 1
...         if self.idx >= len(self.A):
...             raise StopIteration
...         return self.A[self.idx]
...
>>> a = A(10)
>>> for i in a: print i
```