Generators

- Any procedure or method with a `yield` statement is called a **generator**

```
def genTest():
    yield 1
    yield 2
```

- `genTest()` ➔ `<generator object genTest at 0x201b878>`
- Generators have a next() method which starts/resumes execution of the procedure. Inside of generator:
  - `yield` suspends execution and returns a value
  - Returning from a generator raises a `StopIteration` exception

Using a generator

```
>>> foo = genTest()

>>> foo.next()
1
```

Execution will proceed in body of foo, until reaches first yield statement; then returns value associated with that statement

```
>>> foo.next()
2

>>> foo.next()
Results in a StopIteration exception
```

Execution will resume in body of foo at point where stop, until reaches next yield statement; then returns value associated with that statement

Using generators

- We can use a generator inside a looping structure, as it will continue until it gets a StopIteration exception:

```
>>> for n in genTest():
        print n


1
2
>>>
```

A fancier example:

```
def genFib():
    fibn_1 = 1 #fib(n-1)
    fibn_2 = 0 #fib(n-2)
    while True:
        # fib(n) = fib(n-1) + fib(n-2)
        next = fibn_1 + fibn_2
        yield next
        fibn_2 = fibn_1
        fibn_1 = next
```

A fancier example

- Evaluating

```
fib = genFib()
```

- creates a generator object
- Calling

```
fib.next()
```

- will return the first Fibonacci number, and subsequence calls will generate each number in sequence
- Evaluating

```
for n in genFib():
    print n
```

- will produce all of the Fibonacci numbers (an infinite sequence)

Why generators?

- A generator separates the concept of computing a very long sequence of objects, from the actual process of computing them explicitly
- Allows one to generate each new objects as needed as part of another computation (rather than computing a very long sequence, only to throw most of it away while you do something on an element, then repeating the process)

Fix to Grades class

Before

```
def allStudents(self):
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    return self.students[:]
    #return copy of list of students
```

After

```
def allStudents(self):
    if not self.isSorted:
        self.students.sort()
        self.isSorted = True
    for s in self.students:
        yield s
```