MAKING AN ITERABLE FROM A GENERATOR

Generators become exhausted

Generator functions are functions that use yield

A generator function is a generator factory → they return a (new) generator when called

Generators are iterators

→ they can become exhausted (consumed)

→ they cannot be "restarted"

This can lead to bugs if you try to iterate twice over a generator

Example

```
def squares(n):
    for i in range(n):
        yield i ** 2
```

```
sq = squares(5)
```

→ `sq` is a new generator (iterator)

```
l = list(sq)
```

`l` → `[0, 1, 4, 9, 16]`

and `sq` has been exhausted

```
l = list(sq)
```

`l` → `[]`

Example

This of course can lead to unexpected behavior sometimes...

```
def squares(n):
    for i in range(n):
        yield i ** 2
```

```
sq = squares(5)
```

```
enum1 = enumerate(sq)
```

`enumerate` is lazy → hasn't iterated through `sq` yet

```
next(sq)    → 0
next(sq)    → 1
```

```
list(enum1) → [(0,4), (1, 9), (2, 16)]
```

notice how enumerate started at `i=2`

and the index value returned by enumerate is `0`, not `2`

Making an Iterable

This behavior is no different than with any other iterator

As we saw before, the solution is to create an iterable that returns a new iterator every time

```
def squares(n):
    for i in range(n):
        yield i ** 2
```

```
class Squares:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        return squares(n)
```

new instance of the generator

```
sq = Squares(n)

l1 = list(sq)      l1 → [0, 1, 4, 9, 16]

l2 = list(sq)      l2 → [0, 1, 4, 9, 16]
```

Code Exercises