ITERATORS

Where we're at so far...

We created a custom container type object with a `__next__` method

But it had several drawbacks:

- → cannot use a `for` loop
- → once we start using next there's no going back
- → once we have reached `StopIteration` we're basically done with the object

Let's tackle the `loop` issue first

We saw how to iterate using `__next__`, `StopIteration`, and a `while` loop

This is actually how Python handles `for` loops in general

Somehow, we need to tell Python that our class has that `__next__` method and that it will behave in a way consistent with using a `while` loop to iterate

Python knows we have `__next__`, but how does it know we implement `StopIteration`?

The iterator Protocol

A protocol is simply a fancy way of saying that our class is going to implement certain functionality that Python can count on

To let Python know our class can be iterated over using `__next__` we implement the iterator protocol

The iterator protocol is quite simple – the class needs to implement two methods:

→ `__iter__` this method should just return the object (class instance) itself

sounds weird, but we'll understand why later

→ `__next__` this method is responsible for handing back the next element from the collection and raising the `StopIteration` exception when all elements have been handed out

An object that implements these two methods is called an iterator

Iterators

An iterator is therefore an object that implements:

`__iter__` → just returns the object itself

`__next__` → returns the next item from the container, or raises `StopIteration`

If an object is an iterator, we can use it with `for` loops, comprehensions, etc

Python will know how to loop (iterate) over such an object
(basically using the same `while` loop technique we used)

Example

Let's go back to our `Squares` example, and make it into an iterator

```
class Squares:
    def __init__(self, length):
        self.i = 0
        self.length = length

    def __next__(self):
        if self.i >= self.length:
            raise StopIteration
        else:
            result = self.i ** 2
            self.i += 1
            return result


    def __iter__(self):
        return self
```

```
sq = Squares(5)

for item in sq:
    print(item)
```

→

```
0
1
4
9
16
```

Still one issue though!

The iterator cannot be "restarted"

Once we have looped through all the items

the iterator has been exhausted

To loop a second time through the collection we have to create a new instance and loop through that

Code Excercises