ITERATING CALLABLES

Iterating over the return values of a callable

Consider a callable that provides a countdown from some start value:

```
countdown() → 5
countdown() → 4
countdown() → 3
countdown() → 2
countdown() → 1
countdown() → 0
countdown() → -1
...
```

We now want to run a loop that will call `countdown()` until `0` is reached

We could certainly do that using a loop and testing the value to break out of the loop once `0` has been reached

```
while True:
    val = countdown()
    if val == 0:
        break
    else:
        print(val)
```

An iterator approach

We could take a different approach, using iterators, and we can also make it quite generic

Make an iterator that knows two things:

- the callable that needs to be called
- a value (the sentinel) that will result in a `StopIteration` if the callable returns that value

The iterator would then be implemented as follows:

when `next()` is called:

- call the callable and get the result
- if the result is equal to the sentinel → `StopIteration`
  and "exhaust" the iterator
- otherwise return the result

We can then simply iterate over the iterator until it is exhausted

The first form of the `iter()` function

We just studied the first form of the `iter()` function:

`iter(iterable)` → iterator for iterable

if the iterable did not implement the iterator protocol, but implemented the sequence protocol

`iter()` creates a iterator for us (leveraging the sequence protocol)

Notice that the `iter()` function was able to generate an iterator for us automatically

The second form of the `iter()` function

```
iter(callable, sentinel)
```

This will return an iterator that will:

call the callable when next() is called

and either raise `StopIteration` if the result is equal to the sentinel value

or return the result otherwise

Coding Exercises