ITERATING COLLECTIONS

Iterating Sequences

We saw that in the last section → `__getitem__`

→ assumed indexing started at `0`

→ iteration: `__getitem__(0)`, `__getitem__(1)`, etc

But iteration can be more general than based on sequential indexing

All we need is:

a bucket of items → collection, container

get next item → no concept of ordering needed

→ just a way to get items out of the container one by one

a specific order in which this happens is not required – but can be

Example: Sets

Sets are unordered collections of items

```
s = {'x', 'y', 'b', 'c', 'a'}
```

Sets are not indexable

```
s[0]
```

→ `TypeError – 'set' object does not support indexing`

But sets are iterable

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

→

```
y
c
x
b
a
```

Note that we have no idea of the order in which the elements are returned in the iteration

The concept of next

For general iteration, all we really need is the concept of "get the next item" in the collection

If a collection object implements a `get_next_item` method

we can get elements out of the collection, one after the other, this way:

```
get_next_item()
get_next_item()
get_next_item()
```

and we could iterate over the collection as follows:

```
for _ in range(10):
    item = coll.get_next_item()
    print(item)
```

But how do we know when to stop asking for the next item?

i.e. when all the elements of the collection have been returned by calling `get_next_item()`?

→ `StopIteration` built-in Exception

Attempting to build an Iterable ourselves

Let's try building our own class, which will be a collection of squares of integers

We could make this a sequence, but we want to avoid the concept of indexing

In order to implement a next method, we need to know what we've already "handed out"

so we can hand out the "next" item without repeating ourselves

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

    def next_(self):
        result = self.i ** 2
        self.i += 1
        return result
```

Iterating over Squares

```
sq = Squares()


for _ in range(5):
    item = sq.next_()
    print(item)
```

→

```
0
1
4
9
16
```

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

    def next_(self):
        result = self.i ** 2
        self.i += 1
        return result
```

There are a few issues:

→ the collection is essentially infinite

→ cannot use a `for` loop, comprehension, etc

→ we cannot restart the iteration "from the beginning"

Refining the Squares Class

we first tackle the idea of making the collection finite

- we specify the size of the collection when we create the instance
- we raise a `StopIteration` exception if `next_` has been called too many times

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

    def next_(self):
        result = self.i ** 2
        self.i += 1
        return result
```

```
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
```

Iterating over Squares instances

```
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
```

```
sq = Squares(5)          create a collection of length 5

while True:              start an infinite loop

    try:
        item = sq.next_()    try getting the next item
        print(item)

    except StopIteration:    catch the StopIteration exception → nothing left to iterate
        break                break out of the infinite while loop – we're done iterating
```

Output:

```
0
1
4
9
16
```

Python's next( ) function

Remember Python's len( ) function?

We could implement that function for our custom type by implementing the special method: __len__

Python has a built-in function: next( )

We can implement that function for our custom type by implementing the special method: __next__

```
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
```

Iterating over Squares instances

```
sq = Squares(5)
while True:
    try:
        item = next(sq)
        print(item)
    except StopIteration:
        break
```

Output:

```
0
1
4
9
16
```

We still have some issues:

- cannot iterate using `for` loops, comprehensions, etc
- once the iteration starts we have no way of re-starting it
  - and once all the items have been iterated (using next) the object becomes useless for iteration → exhausted

Code Exercises