ITERATORS AND ITERABLES

Iterators

We saw than an iterator is an object that implements

`__iter__` → returns the object itself

`__next__` → returns the next element

The drawback is that iterators get exhausted → become useless for iterating again

→ become throw away objects

But two distinct things going on:

maintaining the collection of items (the container) (e.g. creating, mutating (if mutable), etc)

iterating over the collection

Why should we have to re-create the collection of items just to iterate over them?

Separating the Collection from the Iterator

Instead, we would prefer to separate these two

Maintaining the data of the collection should be one object

Iterating over the data should be a separate object → iterator

That object is throw-away → but we don't throw away the collection

The collection is iterable

but the iterator is responsible for iterating over the collection

The iterable is created once

The iterator is created every time we need to start a fresh iteration

Example

```
class Cities:
    def __init__(self):
        self._cities = ['Paris', 'Berlin', 'Rome', 'London']
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index >= len(self._cities):
            raise StopIteration
        else:
            item = self._cities[self._index]
            self._index += 1
            return item
```

`Cities` instances are iterators

Every time we want to run a new loop, we have to create a new instance of Cities

This is wasteful, because we should not have to re-create the `_cities` list every time

Example So, let's separate the object that maintains the cities, from the iterator itself

```
class Cities:
    def __init__(self):
        self._cities = ['New York', 'New Delhi', 'Newcastle']

    def __len__(self):
        return len(self._cities)


class CityIterator:

    def __init__(self, cities):
        self._cities = cities
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index >= len(self._cities):
            raise StopIteration
        else:
            etc…
```

Example

To use the `Cities` and `CityIterator` together here's how we would proceed:

```
cities = Cities()
```

create an instance of the container object

```
city_iterator = CityIterator(cities)
```

create a new iterator – but see how we pass in the existing `cities` instance

```
for city in cities_iterator:
    print(city)
```

can now use the iterator to iterate

At this point, the `cities_iterator` is exhausted

If we want to re-iterate over the collection, we need to create a new one

```
city_iterator = CityIterator(cities)

for city in cities_iterator:
    print(city)
```

But this time, we did not have to re-create the collection – we just passed in the existing one!

So far…

At this point we have:

a container that maintains the collection items

a separate object, the iterator, used to iterate over the collection

So we can iterate over the collection as many times as we want

we just have to remember to create a new iterator every time

It would be nice if we did not have to do that manually every time

and if we could just iterate over the `Cities` object instead of `CityIterator`

This is where the formal definition of a Python iterable comes in…

Iterables

An iterable is a Python object that implements the iterable protocol

The iterable protocol requires that the object implement a single method

`__iter__` returns a new instance of the iterator object used to iterate over the iterable

```
class Cities:
    def __init__(self):
        self._cities = ['New York', 'New Delhi', 'Newcastle']

    def __len__(self):
        return len(self._cities)


    def __iter__(self):
            return CityIterator(self)
```

Iterable vs Iterator

An iterable is an object that implements

`__iter__` → returns an iterator (in general, a new instance)

An iterator is an object that implements

`__iter__` → returns itself (an iterator) (not a new instance)

`__next__` → returns the next element

So iterators are themselves iterables

but they are iterables that become exhausted

Iterables on the other hand never become exhausted

because they always return a new iterator that is then used to iterate

Iterating over an iterable

Python has a built-in function `iter()`

It calls the `__iter__` method (we'll actually come back to this for sequences!)

The first thing Python does when we try to iterate over an object

it calls `iter()` to obtain an iterator

then it starts iterating (using `next`, `StopIteration`, etc)

using the iterator returned by `iter()`

Code Exercises