`Iterator vs` `iterable`

The Similarity

Even before we understand difference, let's understand similarities.

- **An `Iterable` is basically an object that any user can iterate over**
- An `Iterator` is also an object that any user can iterate over
- In simple words, we can iterate on both `iterables` and `iterators` using `for-loop`.

The difference

- Every `iterator` is an `iterable` but not vice versa. For example, `list` is an `iterable` but not an `iterator`.
- `iterables` can be converted into `iterator` using in-built function called `iter()` .

Example -

```
>>>
>>> cities = ["Delhi", "Pune", "Mumbai"]  # <-- `cities` is an iterable
>>> c_iterator = iter(cities)  # <-- We converted `iterable` into `iterator`
>>>
>>> type(c_iterator)  # <-- You can see converted data type here.
<class 'list_iterator'>
>>>
>>> next(c_iterator)  # <-- Each `next()` call gives 1 element (Lazy evaluation)
'Delhi'
>>>
>>> next(c_iterator)
'Pune'
```

```
>>>
>>> next(c_iterator)
'Mumbai'
>>>
>>> next(c_iterator)  # <-- When iterator is exhausted `StopIteration` is raised.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
>>>
```

Iterator protocol

- `iterator` is an object having both `__iter__` and `__next__` methods.
- The `__next__()` method will be called when in-built function `next()` is applied on an `iterator`

Python iterator objects are required to support two methods while following the iterator protocol.

- `__iter__` returns the iterator object itself. This is used in *for* and *in* statements.
- `__next__` method returns the next value from the iterator. If there is no more items to return then it should raise *StopIteration* exception.

First example -

```
"""
-------------------------------------------------------------------------
* PROBLEM STATEMENT *
-------------------------------------------------------------------------

- Write a program to get random 15 natural numbers within range of 1 to 82.
- Use in-build module called `random` to produce random numbers.
- The implementation should be done using class-based iterator

- NOTE :: Follow `iterator` protocol and define `__iter__()` and `__next__()`
-------------------------------------------------------------------------
"""
```

```
import random


class MyIterable:
    def __init__(self):
        self.x = 0
    def __iter__(self):
        return self

    def __next__(self):
        if self.x < 15:
            self.x += 1
            return random.randint(1, 83)
        else:
            raise StopIteration


obj = MyIterable()

for element in obj:
    print(element)
```

Second example -

```
"""
-------------------------------------------------------------------------------
* PROBLEM STATEMENT *
-------------------------------------------------------------------------------

- Write a program to loop over a given sequence in backward direction.
- For example, if the input sequence is [100, 200, 300]
- Output sequence should be :: [300, 200, 100]

- NOTE :: Follow `iterator` protocol and define `__iter__()` and `__next__()`
-------------------------------------------------------------------------------
"""


class Reverse:
    """
    Creates Iterators for looping over a sequence backwards.
    """

    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
```

```
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1

        return self.data[self.index]


sequence = [400, 500, 600]
sequence_backwards = Reverse(sequence)

for element in sequence_backwards:
    print(element)
```

What exactly happens behind the scenes when a `for loop` is executed?

- The `for` statement calls `iter()` on the object and converts it into a `iterator`.
- When object is converted into `iterator`, it will have a `__next__()` method implicitly available.
- The `in` operator that we use in `for-loop` is responsible for calling `next()` (which in turn calls `__next__()`.
- `__next__()` would terminate as soon as `StopIteration` exception is raised (when all elements from the collection are iterated).
- `for-loop` will terminate as soon as it catches `StopIteration` exception.

Third example -

```
class Counter(object):
    def __init__(self, lower, upper):
        self.current = lower
        self.high = high
```

```
    def __iter__(self):
        'Returns itself as an iterator object'
        return self

    def __next__(self):
        'Returns the next value till current is lower than high'
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

```
>>> c = Counter(5,10)
>>> for i in c:
...   print(i, end=' ')
...
```

NOTE

Remember that an iterator object can be used only once. It means after it raises `StopIteration` once, it will keep raising the same exception.

What are Iterators?

- Iterator is an abstraction.
- Iterators enable programmers to use access elements of an iterable object (list, set, string, tuple etc) without any deeper knowledge of the data structure of this object.
- `for-loops` use `itarators` implicitly.

Why iterators?

- **Saving memory space**
- Lazy evaluation

- When you’re working with lots of data (If you don’t want to load all the data in memory at a time)