Python Iterators Concept:

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().
- Iterator in python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets.
- The iterator object is initialized using the iter() method. It uses the next() method for iteration.

__iter(iterable)__

This method is called for the initialization of an iterator. This returns an iterator object.

next (__next__ in Python 3) :

The next method returns the next value for the iterable.

When we use a for loop to traverse any iterable object, internally it uses the **iter()** method to get an iterator object which further uses next() method to iterate over.

This method raises a **StopIteration** to signal the end of the iteration.

Iterator vs Iterable :

Lists, tuples, dictionaries, and sets are all **iterable** objects. They are iterable containers which you can get an iterator from.

All these objects have a **iter()** method which is used to get an **iterator**.

Q. Return an iterator from a tuple, and print each value.?

```
mytuple = ("apple" , "banana" , "cherry")
myiter = iter(mytuple)
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:

apple
banana
cherry

Even strings are iterable objects, and can return an iterator.

```
mystr = "banana"
myiter = iter(mystr)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:

b
a
n
a
n
a

Looping Through an Iterator:

We can also use a **for loop** to iterate through an iterable object:

Example:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

Output:

apple
banana
cherry

------>> Iterate the characters of a string:

Example:

```
mystr = "banana"
for x in mystr:
  print(x)
```

Note : **The for loop actually creates an iterator object and executes the next() method for each loop.**

Create an Iterator:

- To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.
- As you have learned in the Python Classes/Objects chapter, all classes have a function called __init__(), which allows you to do some initializing when the object is being created.
- The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
- The __next__() method also allows you to do operations, and must return the next item in the sequence.

Q. Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x

myclass = MyNumbers()
myiter = iter(myclass)
```

```
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:

```
1
2
3
4
5
```

StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration to go on forever, we can use the **StopIteration** statement.

In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example: WAP to display numbers and Stop after 5 iterations:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 5:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration
```

```
myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
  print(x)
```

Output:

```
1 2 3 4 5
```

******** ====== ******* ====== *********