KUNAL SIR

ITERABLES & ITERATORS

ITERABLES:

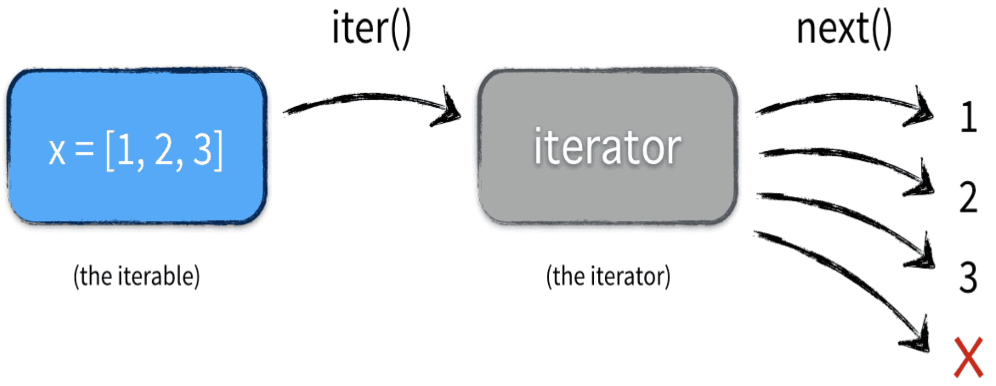
- Iterable is an object, which one can iterate over.
- All Data Structures like list, tuple, set and other things like files, string etc are all iterable class.
- If we try to iterate iterable object then it will give error.
- We need an iterator object to iterate iterable object.
- We can get an Iterator object when iterable is passed to iter() method.

ITERATORS:

- Iterator is one type of protocol.
- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
- Technically speaking, a Python iterator object must implement two special methods, __iter__() and __next__(), collectively called the iterator protocol.
- Iterator object is used for getting elements sequence wise using __next__() method.
- Every Iterator object can be iterable object but every Iterable object is not an iterator object.
- Iterator is used to get finite and infinite sequence of elements.

e.g

```
>>> l = [1,2,3]
>>> x = iter(l)   #x = Iterator object, l = iterable object
>>> next(x) #1
>>> next(x) #2
>>> next(x) #3
>>> next(x) #StopIteration Error
```



- If class implements the two methods __iter__() and __next__() then class object becomes an Iterator object.
- We can implement iterator and iterable in same class.

  e.g. class A:

  ```
          __iter__()
          __next__()
  ```

- We can implement separate iterable class.
  e.g. class A:

  ```
      __iter__()
  ```

- We can create separate iterator class but we can make it as iterable also.

  ```
  e.g. class B:          →          class B:
          __next__()                       __iter__()
                                           __next__()
  ```

Class containing both Iterable and Iterator.

```
class Myrange:
  def __init__(self,n):
    self.n = n
    self.i = 0

  def __iter__(self):
    return self

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

OUTPUT:

```
>>> x = Myrange(3)
>>> next(x)
0
>>> next(x)
1
>>> next(x)
2
>>> next(x)
StopIteration
```

Iterable & Iterator in separate class.

```
class Myrange:     #Iterable class
  def __init__(self,n):
    self.n = n

  def __iter__(self):
    return Myrange_itr(self.n)


class Myrange_itr:    #Iterator class
  def __init__(self,n):
    self.n = n
```

```
        self.i = 0

    def __iter__(self):
        return self

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

OUTPUT:

```
>>> x = Myrange(3)
>>> y = iter(x)
>>> next(y)
0
>>> next(y)
1
>>> next(y)
2
>>> next(y)
 StopIteration
>>> x = Myrange_itr(3)
>>> next(x)
0
>>> next(x)
1
>>> next(x)
2
```

GENERATORS

- It is a simple way to implement iterator in python.
- A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the **yield** keyword rather than return.
- If the body of a def contains at least one yield, the function automatically becomes a generator function.
- Generator functions return a generator object.
- Generator objects are used either by calling the next method on the generator object or using the generator object in a for loop.
- There is a lot of work in building an iterator class in Python. We have to implement a class with __iter__() and __next__() method, keep track of current states, and raise StopIteration when there are no values to be returned.
- Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Differences between Generator function and Normal function

- Generator function contains one or more yield statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

e.g.

```
1) def m1():
           yield ‘Java’
           yield ‘Python’
           yield ‘Testing’
  itr = m1()  #itr = iterator object
  print(next(itr)) #Java
  print(next(itr)) #Python
  print(next(itr)) #Testing
```

```
2) def myrange(n):
           i = 0
           while i < n :
                     yield i
                     i = i + 1
  print(“Iterating using for loop:”)
  for x in myrange(3):
           print(x)
  print(“Iterating using next() method:”)
  itr = myrange(3)
  print(next(itr))
  print(next(itr))
  print(next(itr))
```

OUTPUT:

```
Iterating using for loop:
0
1
2
Iterating using next() method:
0
1
2
```