Iterables V/s Iterators

An iterable is an object that contains a countable number of values, It cab be iterated upon, meaning that you can traverse through all the values. Such objects basically implemets __iter()__ method. Since we can iterate through list hence LIST is iterable. All the items of iterables are allocated in the memory locations. Lists, tuples, dictionaries, sets, ans strings 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.

Iterator can implement __iter__() or iter(), and __next__() or next()

In [1]:
```
lst=[1,2,3,4,5,6,7,8,9,0]
for i in lst:
    print(i)
```

```
1
2
3
4
5
6
7
8
9
0
```

iter() will convert iterables into iterator

Properties of ITERATOR:

```
0. It is used when we are required to take consideration of space complexity. Because all the
elements of the iterators will not get initialized at once. If we are required to get elements one
by one, then we use iterator, and hence with the help of next() function we access the element one
by one.
1. Unlike iterables, where all the values of it are initialized in the memory when it is created,
iterators are not initialized in the memory when they are created.
```

2. When the in-built next(iterator) function is called, then only first value of the iterator is initialized in the memory.
3. When the elements are exhausted from the iterartors, next() function will throw an error.
4. To start from first element again, again make a iterator variable and then use next() function.

In [2]:
```
lst=[1,2,3,4,5,6,7,8,9,0]
print(iter(lst))
lst1 = iter(lst)
next(lst1)
```

```
<list_iterator object at 0x7f325422fa90>
```

Out[2]: 1

In [3]:
```
next(lst1)
```

Out[3]: 2

In [4]:
```
next(lst1)
```

Out[4]: 3

1. Another way of accessing the iterator elements is by using a loop statement, e.g. for loop. In for loop, StopIteration Exception is implicitly handeled. That is why, it is not throwing an error.

In [5]:
```
lst=[1,2,3,4,5,6,7,8,9,0]
lst1 = iter(lst)

for i in lst1:
    print(i)
```

```
1
2
3
4
5
6
7
```

```
8
9
0
```

GENERATORS

They are created because they are used to create disguised iterators itself. Basically, we perform the working of iterator, but with the generator technique. At the end, not iterator,but generator is created.

1. To create iteration, we use **iter()** keyword.
2. To create generator, we use **yield** keyword, along with function where yield will be in place of **return**.
3. **yield** saves the local variable value. It also returns the local variable value.
4. Once iterator/generator is created, we use **next()** to access the value.
5. They helps to write fast and compact code.
6. Performance of generators are better than iterators.
7. Iterators are much more memory efficient.
8. Generators are derived from iterator class itself.

In [6]:
```
def square(n):
    for i in range(n):
        yield i**2
```

In [7]:
```
square(3)
```

Out[7]:
```
<generator object square at 0x7f32542482e0>
```

In [8]:
```
for i in square(3):
    print(i)
```

```
0
1
4
```

In [9]:
```
a=square(3)

while True:
    try:
        print(next(a))
    except StopIteration:
        print("Itne mei itna hi milega")
        break
```

```
0
1
4
Itne mei itna hi milega
```

In [10]:
```
def even_gen():
    n=0

    n+=2
    yield n

    n+=2
    yield n

    n+=2
    yield n

num = even_gen()
print(next(num))
print(next(num))
print(next(num))
print(next(num))
```

```
2
4
6
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-10-5b7c6dd7a48a> in <module>
     15 print(next(num))
     16 print(next(num))
```

```
---> 17 print(next(num))
```

In [11]:

```
def fibo():
    n1=0
    n2=1
    while True:
        yield n1
        n1,n2=n2,n1+n2

seq=fibo()

print(next(seq))
print(next(seq))
print(next(seq))
print(next(seq))
print(next(seq))
print(next(seq))
print(next(seq))
# as many number of times because we are using generators, and the StopIteration exception is not present hee like i
```

```
0
1
1
2
3
5
8
```

In [12]:

```
import types, collections

issubclass(types.GeneratorType, collections.Iterator)
```

```
<ipython-input-12-774d58af0d44>:3: DeprecationWarning: Using or importing the ABCs from 'collections' instead of fro
m 'collections.abc' is deprecated since Python 3.3, and in 3.9 it will stop working
  issubclass(types.GeneratorType, collections.Iterator)
```

Out[12]: True