Generators and Iterators

Generator:

if a function which in turn calls yield one or more times returns an object called the generator. No statement in the function is executed when called if the function contains yield statement.

The code is taken from the file 1_gen.py.

```
def mygen():
    print("one")
    yield 10
    print("two")
    yield 20
    print("three")
    yield 30
    print("four")
f = mygen()
```

f is a generator object.

A generator object is iterable.

So, we can call next on the iterable object.

```
res = next(f) # one
print(res) # 10
```

When next(f) is called, the function mygen executes until and inclusive of the yield statement. The yield returns the control to the called of next(gen object) and returns the value of the expression in the yield statement. The generator function remembers the position of the statement following the yield. The control is transferred when the next is called on this generator object again.

The statements below will cause the statements in the generator function to be executed from the position of the last yield till the next yield

```
res = next(f) # two
print(res) # 20
```

This goes on until the end of the generator function is reached. At that point the generator function throws the exception stop iteration.

As the generator object is iterable, it can be used in for loops.

Observe the following about generators.
a) The generator function has one or more yields
b) The generator function as well the code calling next stay in some state of execution simultaneously. This concept is called co-routine.
c) The generator function does not execute first time it is called – instead returns a generator object.
d) The generator function resumes from where it had left off in the earlier execution of call on the generator object
e) The generator objects are iterable
f) The generators are lazy – like me!. They do not produce all the results at a time.

Let us have a look at an infinite sequence generator from the file 2_gen.py.

The function is_prime is a helper function which checks whether a given number is prime – some stupid function this!

```
def is_prime(m):
        i = 2
        while m % i != 0 :
                i += 1
        return i == m
```

This function is a generator function – returns a generator object when it is called. Each time next is called it returns a number. The first time it returns 2 – then 3 – then starts from odd number 5. If the number is odd it yields it. Each time next is called, it examines whether the next odd number is prime and yields only prime numbers. This way this generator can generate an infinite sequence of prime numbers.

```
def gen():
```

```
        yield 2
        yield 3
        m = 5
        while True:
                if is_prime(m) :
                        yield m
                m += 2


g = gen()
# get next n primes
n = 25
for i in range(n):
        print(next(g))
```

Iterators:

In an examination hall, there could be students writing different examinations of different subjects – say the number of subjects is 4. How should the invigilator distribute the papers? Should he distribute in the order in which the students sit? Should he distribute in the order of subjects? Can we have more than one invigilator?

The class room has number of students – it is a container or a collection. The invigilator has to visit each student. He is iterating through the students in the class. He is not necessarily part of the class. He is an iterator. It is possible to have a class with multiple invigilators. A container can have multiple iterators. Observe the fact that walking though a container depends on a container, but is not part of the container normally. This concept is a very important concept.

The two terms I want you to master from this course are **interface** and **implementation**. We have observed that a for statement can walk through a container provided it is iterable. What does that mean?

The container class (like list) should support a function __iter__ (callable as iter(container_object)) which returns an object of a class – the object is called an iterator. This class should support __next__(callable as next(iterator_object)). These should functions are interfaces. We can implement any way we want to support walking though a container logically. For example, we may visit only elements in odd position or elements satisfying a boolean condition – like elements greater than 100.

Let us look at a couple of examples.

This is from the file 3_iter.py

```
class MyContainer:
        def __init__(self, mylist):
                self.mylist = mylist

        def __iter__(self):
                self.i = 0
                return self

        def __next__(self):
                self.i += 1
                if self.i <= len(self.mylist):
                        return self.mylist[self.i - 1]
                else:
                        raise StopIteration

a = [ 'apple', 'banana', 'carrot', 'date', 'eff', 'fish' ]
c = MyContainer(a)
for w in c :
        print(w)
```

observe:

c = MyContainer(a)

Creates an object of MyContainer whose attribute mylist refers to the list a.

The for statement will call iter(c). This call is changed to MyContainer.__iter__(c).
This __iter__ function adds a position attribute i to the object. Then it returns the MyContainer object itself as the iterator object.
Then the for statement keeps calling next on this iterable object. This __next__ function has the logic to return the next element from the list and update the position and also raise the exception stop iteration when the end of the list is reached.

But there is one catch in this implementation. Let us examine the following code from the same file.
We create two iterator objects it1 and it2 from the same container object a.
All these refer to the same object. Even though we have two iterator objects, both share the same location index. Calling next on it1 also affects the position index on it2.

```
a = [ 'apple', 'banana', 'carrot', 'date', 'eff', 'fish' ]
c = MyContainer(a)
it1 = iter(c)
it2 = iter(c)
print(next(it1)) # apple
print(next(it2)) # expect apple, we will get banana
```

Let us look at the next example which avoid this. We do that by having another class and making the position index part of the iterator object and not the container object.
The code is from the file : 4_iter.py

```
class MyIterator:
        def __init__(self, c):
                self.c = c
                self.i = 0
        def __next__(self):
                self.i += 1
                if self.i <= len(self.c.mylist):
                        return self.c.mylist[self.i - 1]
```

```
            else:
                raise StopIteration

class MyContainer:
    def __init__(self, mylist):
        self.mylist = mylist

    def __iter__(self):
        return MyIterator(self)



a = [ 'apple', 'banana', 'carrot', 'date', 'eff', 'fish' ]
c = MyContainer(a)
for w in c :
    print(w)



a = [ 'apple', 'banana', 'carrot', 'date', 'eff', 'fish' ]
c = MyContainer(a)
it1 = iter(c)
it2 = iter(c)
print(next(it1)) # apple
print(next(it2)) # we will get banana as expected
```

Observe the changes.
iter function in MyContainer class returns an object of MyIterator class.
The iterator object holds a reference to the MyContainer object and also holds the position index.
it1 = iter(c)
it2 = iter(c)
These are two different objects having their own position index.
So, next(it1) does not affect next(it2).

Thats all about generators and iterators as of now.