Experiment No : 04

Aim : Write python programs to understand Decorators, Iterators and Generators.

Description :

Iterators

An iterator is an object that can be iterated upon which means that you can traverse through all the values. List, tuples, dictionaries, and sets are all iterable objects.

Iterators allow us to create and work with lazy iterable which means you can use an iterator for the lazy evaluation. This allows you to get the next element in the list without re-calculating all of the previous elements. Iterators can save us a lot of memory and CPU time.

Generators

Generator functions act just like regular functions with just one difference that they use the Python yieldkeyword instead of return . A generator function is a function that returns an iterator. A generator expression is an expression that returns an iterator. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for in” loop.

Decorator

A decorator in Python is any callable Python object that is used to modify a function or a class. It takes in a function, adds some functionality, and returns it. Decorators are a very powerful and useful tool in Python since it allows programmers to modify/control the behavior of function or class. Decorators are usually called before the definition of a function you want to decorate.

Implementation :

Code :

 # A) ITERATORS

```
#iterable  __iter__() or __getitem__() (python object)

#iterator  __next__()  (python object)

#iteration:


# for loop use for iteration on (list,tuple,dictionary,string ,file )

a=["hey", "bro", "you'r", "aweasome"]

for i in a:

  print(i)

# shows list of methods

print(dir(a))



# gives iterator object

itr=iter(a)

print(itr)

# bring first element

print(next(itr))

# for second element

print(next(itr))
```

```
# for third elemrnt

print(next(itr))

# fourth element

print(next(itr))


#  stop iteration

#next(itr)

# reversed method( built in method)

a=["hey", "bro", "you'r", "aweasome"]

itr=reversed(a)


print(next(itr))

#brings last element(awesome)

print(next(itr))

print(next(itr))

#next(itr)

print(dir(a))


# brings list of methods


# implement remote control class that allows you to

# press "next" button to go to next TV channel.
```

```
class RemoteControl():
    def __init__(self):
        self.channels=["HBO", " CNN", "ABC", "ESPN"]
        self.index = -1
    def __iter__(self):
        return self
    def __next__(self):
        self.index +=1
        if self.index ==len(self.channels):
            raise StopIteration
        return self.channels[self.index]
r=RemoteControl()
itr=iter(r)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
#print(next(itr))
```

 # B) GENERATORS

Generator is a simple way of creating iterator.

```
# Benefits over class based iterator:
# you don't need to define iter() and next() methods
# you don't need to raise Stopiteration exception

def remote_control_next():
    yield "cnn"
    yield "espn"

itr=remote_control_next()
print(itr)
print(next(itr))
print(next(itr))

for c in remote_control_next():
     print(c)

#example generator(generator generates on the fly value)
# yield (on the fly value generate)
def gen(n):
    for i in range(n):
        yield i
         # return i  ( output 0)
```

```
#in oder to save ram memory
g=gen(35454545454545454)
#no need to save this no. in memory
print(g)
g=gen(3)
print(g.__next__())
print(g.__next__())
print(g.__next__())
# only 0 to 2 will print
#print(g.__next__())
#print(g.__next__())
#error


#example


# fibbonaci series 0,1,1,2,3,5,8,13,....
# 0+1=1, 1+1=2, 1+2=3, 2+3=5,
# we are going to produce fibonacci sequence using generator.
def fib():
    a,b = 0,1
    while True:
```

```
        yield a
        a,b = b, a+b
for f in fib():
   if f>100:
      break
   print(f)
```

 # C) DECORATORS

```
# decorators: modify the functionality of function


def function1():
   print("subscribe now")
function2= function1
# if we delete function1
del function1
function2()
#it will run because function2 (copy already created)


# function inside function
def function(num):
   if num==0:
      return print
```

```
    if num==1:
        return sum
a=function(1)
# print output shows we can return function through function
print(a)

# function inside function as an argument.(if we write (sum) instead of(print). it
# will not work.
def executor(function):
    function("this")
executor(print)

# decorators: modify the functionality of function

def decor1(function1):
    def nowexec():
        print("executing now")
        function1()
        print("executed")
    return nowexec
def who_is_anuradha():
    print("anuradha is a teacher")
```

```
who_is_anuradha = decor1(who_is_anuradha)
who_is_anuradha()


# @ decorators
def decor1(function1):
    def nowexec():
        print("executing now")
        function1()
        print("executed")
    return nowexec
@decor1
def who_is_anuradha():
    print("anuradha is a teacher")
#who_is_anuradha = decor1(who_is_anuradha)
who_is_anuradha()
```

Output :

A) Iterators

```
hey
bro
you'r
awesome
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
<list_iterator object at 0x00000252BC3BF220>
hey
bro
you'r
aweasome
aweasome
you'r
bro
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
HBO
 CNN
ABC
ESPN
```

B) Generatores

```
<generator object remote_control_next at 0x0000021BE7DEECF0>
cnn
espn
cnn
espn
<generator object gen at 0x0000021BE7E08CF0>
0
1
2
0
1
1
2
3
5
8
13
21
34
55
89
```

C) Decorators

```
subscribe now
<built-in function sum>
this
executing now
Gaurang is a Student
executed
executing now
anuradha is a teacher
executed
```

Conclusion : Therefore we have successfully implemented python programs to understand Decorators, Iterators and Generators.