Experiment 03

Write at least 2 python programs (for each)to understand the concept of Decorators, Iterators and Generators.

Roll No.	01
Name	Aamir Ansari
Class	D10-A
Subject	Python Lab
LO Mapped	LO1: Understand the structure, syntax, and semantics of the Python language LO2: Interpret advanced data types and functions in python

Aim:

Write at least 2 python programs (for each)to understand the concept of Decorators, Iterators and Generators.

Introduction:

1) Decorators

Decorators belong most probably to the most beautiful and most powerful design possibilities in Python, but at the same time the concept is considered by many as complicated to get into. To be precise, the usage of decorators is very easy, but writing decorators can be complicated, especially if you are not experienced with decorators and some functional programming concepts.
Even though it is the same underlying concept, we have two different kinds of decorators in Python:
Function decorators
Class decorators
A decorator in Python is any callable Python object that is used to modify a function or a class. A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class. The modified functions or classes usually contain calls to the original function "func" or class "C".
We know from our various Python training classes that there are some points in the definitions of decorators, where many beginners get stuck.

First Steps to Decorators
We introduce decorators by repeating some important aspects of functions. First you have to know or remember that function names are references to functions and that we can assign multiple names to the same function:

```
def succ(x):
   return x + 1
successor = succ
successor(10)
# Output:
11
succ(10)
# Output:
11
```

Functions inside Functions

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32

    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

print(temperature(20))
It's 68.0 degrees!
```

The following example is about the factorial function, which we previously defined as follows:

```
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

What happens if someone passes a negative value or a float number to this function? It will never end. You might get the idea to check that as follows:

```
def factorial(n):
    """ calculates the factorial of n,
        n should be an integer and n <= 0 """
    if type(n) == int and n >=0:
        if n == 0:
            return 1
        else:
            return n * factorial(n-1)

    else:
        raise TypeError("n has to be a positive integer or zero")
```

If you call this function with 4 " for example, i.e. factorial (4) ", the first thing that is checked is whether it is my positive integer. In principle, this makes sense. The "problem" now appears in the recursion step. Now factorial (3) " is called. This call and all others also check whether it is a positive whole number. But this is unnecessary: If you subtract the value 1 " from a positive whole number, you get a positive whole number or `` 0 " again. So both well-defined argument values for our function.

With a nested function (local function) one can solve this problem elegantly:

```
def factorial(n):
   """ calculates the factorial of n,
      n should be an integer and n <= 0 """
   def inner_factorial(n):
      if n == 0:
         return 1
      else:
         return n * inner_factorial(n-1)
   if type(n) == int and n >=0:
      return inner_factorial(n)
   else:
      raise TypeError("n should be a positive int or 0")
```

<u>Functions and methods are called callable as they can be called</u>
In fact, any object which implements the special __call__() method is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):
   def inner():
      print("I got decorated")
      func()
   return inner

def ordinary():
   print("I am ordinary")
```

When you run the following codes in shell,

```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
```

```
I got decorated
I am ordinary
```

In the example shown above, make_pretty() is a decorator. In the assignment step:

```
pretty = make_pretty(ordinary)
```

The function ordinary() got decorated and the returned function was given the name pretty.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as,

ordinary = make_pretty(ordinary).

This is a common construct and for this reason, Python has a syntax to simplify this.

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:

```
def divide(a, b):
    return a/b
```

This function has two parameters, a and b. We know it will give an error if we pass in b as 0.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
```

```
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner

@smart_divide
def divide(a, b):
    print(a/b)
```

This new implementation will return None if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner, we can decorate functions that take parameters.

2) Iterators

An iterator is an object that implements the iterator protocol. An iterator protocol is nothing but a specific class in Python which further has the __next()__ method. Which means every time you ask for the next value, an iterator knows how to compute it. It keeps information about the current state of the iterable it is working on. The iterator calls the next value when you call next() on it. An object that uses the __next__() method is ultimately an iterator.

Iterators help to produce cleaner looking code because they allow us to work with infinite sequences without having to reallocate resources for every possible sequence, thus also saving resource space. Python has several built-in objects, which implement the iterator protocol and you must have seen some of these before: lists, tuples, strings, dictionaries and even files. There are also many iterators in Python, all of the itertools functions return iterators. You will see what itertools are later on in this tutorial.

Iterables

According to Vincent Driessen of nvie.com, "an iterable is any object, not necessarily a data structure that can return an iterator". Its main purpose is to return all of its elements. Iterables can represent finite as well as infinite sources of data. An iterable will directly or indirectly define two methods: the __iter__() method, which must return the iterator object and the __next()__ method with the help of the iterator it calls.

Note: Often the iterable classes will implement both __iter__()and __next__() in the same class, and have __iter__() return self, which makes the _iterable_ class both an iterable and its own iterator. It's perfectly fine to return a different object as the iterator, though.

There is a major dissimilarity between what an iterable is and what an iterator is. Here is an example:

```
iterable_value = 'Aamir'
iterable_obj = iter(iterable_value)

while True:
    try:
        # Iterate by calling next
        item = next(iterable_obj)
        print(item)
    except StopIteration:

        # exception will happen when iteration will over
        break
```

Output :

```
A
a
m
i
r
```

Another example would be

```
a_set = {1, 2, 3}
b_iterator = iter(a_set)
next(b_iterator)
type(a_set)
type(b_iterator)
```

In the example, a_set is an iterable (a set) whereas b_iterator is an iterator. They are both different data types in Python.

Wondering how an iterator works internally to produce the next sequence when asked? Let's build an iterator that returns a series of number:

```
class Series(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

n_list = Series(1,10)
print(list(n_list))
```

__iter__ returns the iterator object itself and the __next__ method returns the next value from the iterator. If there are no more items to return then it raises a StopIteration exception.

It is perfectly fine if you cannot write the code for an iterator yourself at this moment, but it is important that you grasp the basic concept behind it. You will see generators later on in the tutorial, which is a much easier way of implementing iterators.

3) Generators

It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a yield statement instead of a return statement.

If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

The difference is that while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.
Here is how a generator function differs from a 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.
Here is an example to illustrate all of the points stated above. We have a generator function named my_gen() with several yield statements.

```
# A simple generator function
def my_gen():
   n = 1
   print('This is printed first')
   # Generator function contains yield statements
   yield n

   n += 1
   print('This is printed second')
   yield n

   n += 1
   print('This is printed at last')
   yield n
```

Generators with a Loop
The above example is of less use and we studied it just to get an idea of what was happening in the background.

Normally, generator functions are implemented with a loop having a suitable terminating condition.

Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yield my_str[i]


# For loop to reverse the string
for char in rev_str("hello"):
    print(char)
```

Output

```
o
l
l
e
h
```

In this example, we have used the range() function to get the index in reverse order using the for loop.

Use of Python Generators

There are several reasons that make generators a powerful implementation.

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2 using an iterator class.

```
class PowTwo:
    def __init__(self, max=0):
        self.n = 0
        self.max = max

    def __iter__(self):
        return self
    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

The above program was lengthy and confusing. Now, let's do the same using a generator function.

```
def PowTwoGen(max=0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Since generators keep track of details automatically, the implementation was concise and much cleaner.

2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

3. Represent Infinite Stream

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2
```

4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def fibonacci_numbers(nums):
    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x

def square(nums):
    for num in nums:
        yield num**2

print(sum(square(fibonacci_numbers(10))))
```

Output

```
4895
```

Results:

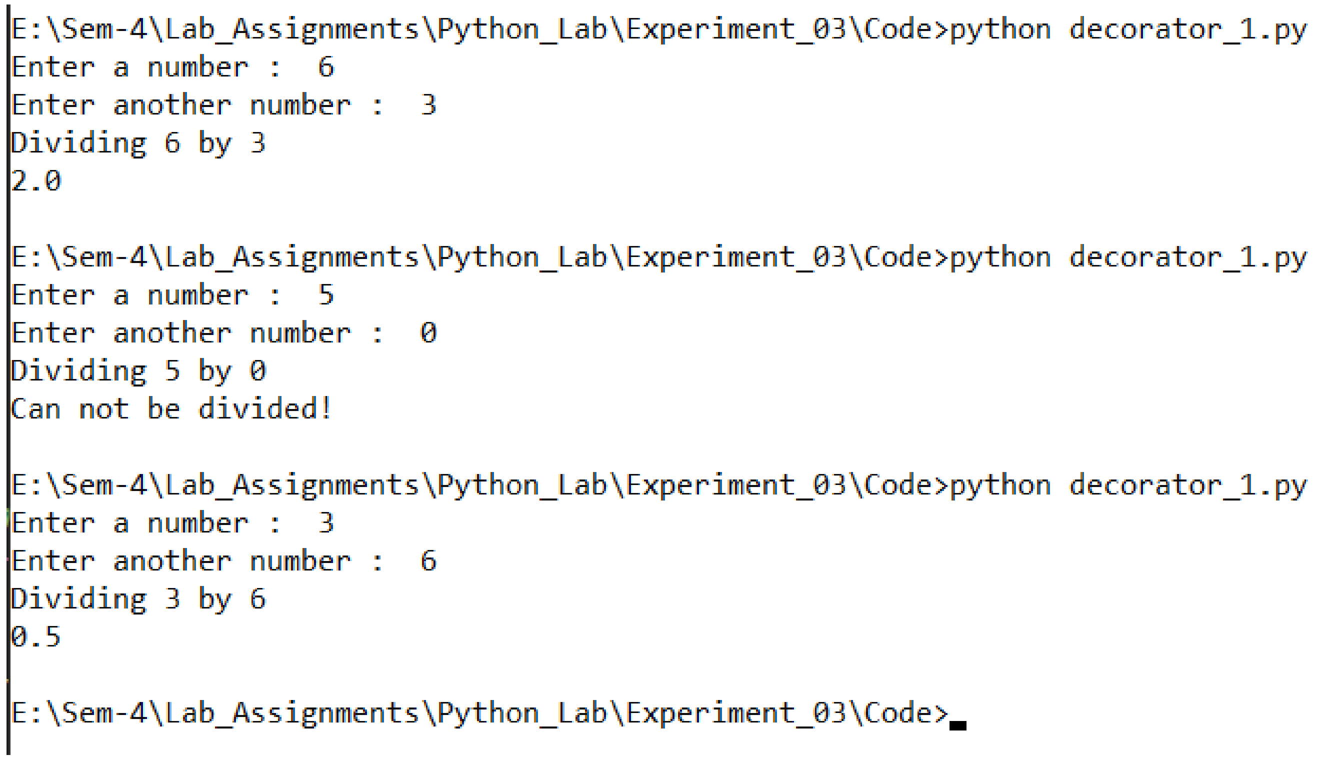
# demonstration of decorators (program 1)

```
def betterDivide(func):
    def innerFunction(a, b):
        print(f"Dividing {a} by {b}")
        if (b == 0):
            print("Can not be divided!")
            return
        return func(a, b)
    return innerFunction

# decorate divide function
def divide(a ,b):
    print(a/b)
divide = betterDivide(divide)

# calling function
divide(3,0)

# output
```



```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python decorator_1.py
Enter a number :  6
Enter another number :  3
Dividing 6 by 3
2.0

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python decorator_1.py
Enter a number :  5
Enter another number :  0
Dividing 5 by 0
Can not be divided!

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python decorator_1.py
Enter a number :  3
Enter another number :  6
Dividing 3 by 6
0.5

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>
```

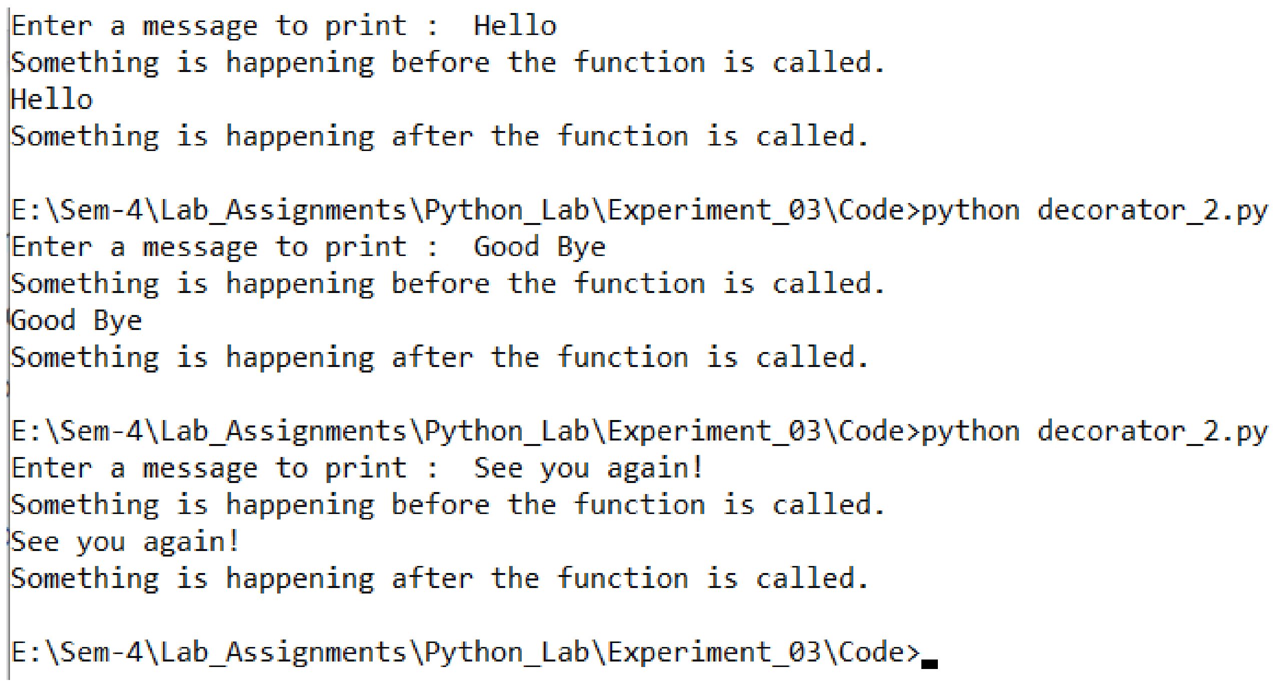
# demonstration of decorators (program 2)

```
def my_decorator(func):
    def wrapper(message):
        print("Something is happening before the function is called.")
        func(message)
        print("Something is happening after the function is called.")
    return wrapper

# syntactic sugar for decorator
@my_decorator
def say_message(message):
    print(message)

msg = input("Enter a message to print :  ")
```

# call the function
say_message(msg)


```
Enter a message to print :  Hello
Something is happening before the function is called.
Hello
Something is happening after the function is called.

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python decorator_2.py
Enter a message to print :  Good Bye
Something is happening before the function is called.
Good Bye
Something is happening after the function is called.

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python decorator_2.py
Enter a message to print :  See you again!
Something is happening before the function is called.
See you again!
Something is happening after the function is called.

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>
```

# Demonstration of iterators (program 1)

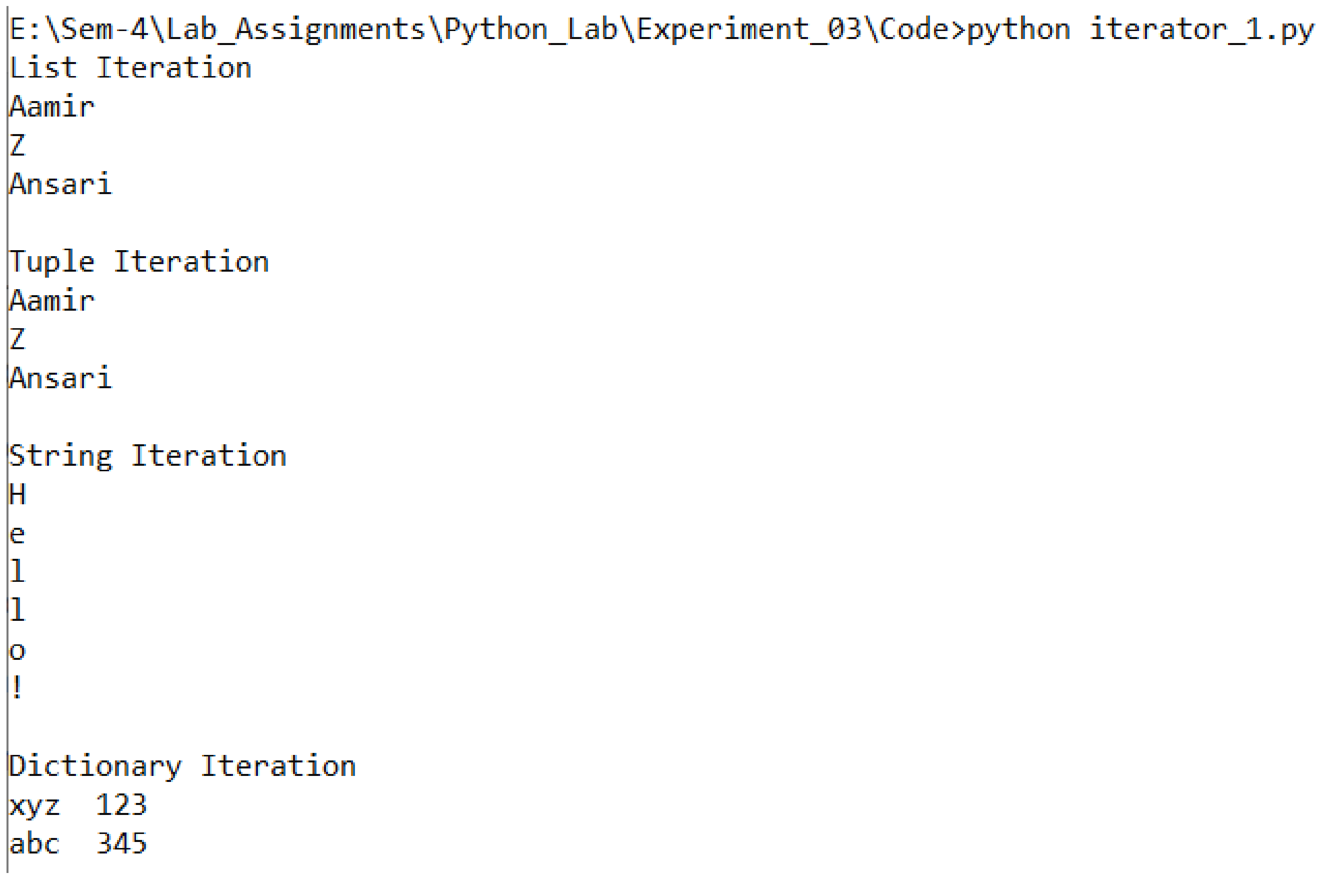
```
# Iterating over a list
print("List Iteration")
l = ["Aamir", "Z", "Ansari"]
for i in l:
   print(i)

# Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("Aamir", "Z", "Ansari")
for i in t:
   print(i)

# Iterating over a String
```

```
print("\nString Iteration")
s = "Hello!"
for i in s :
    print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
    print("%s  %d" %(i, d[i]))
```



```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python iterator_1.py
List Iteration
Aamir
Z
Ansari

Tuple Iteration
Aamir
Z
Ansari

String Iteration
H
e
l
l
o
!

Dictionary Iteration
xyz  123
abc  345
```

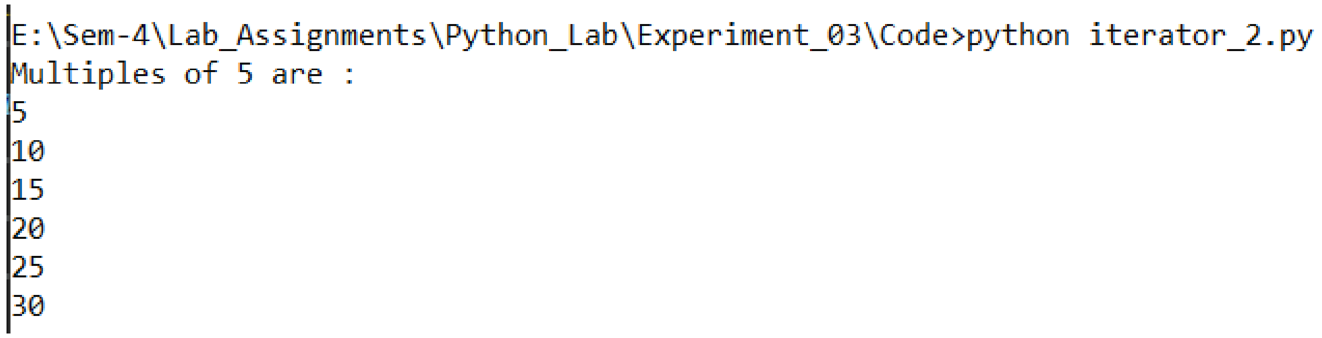
# Demonstration of iterators (program 2)

```
# demonstration of iterators using __iter__ function

multiplesOfFive = [5,10,15,20,25,30]
```

```
x = iter(multiplesOfFive)
# x becomes an itertor of list "multiplesOfFive"

# we iterate through list "multiplesOfFive" using iterator "x" and print
print(“Multiples of 5 are :  \n”)
while True:
    try:
        # next function to move to next element of the object (here, list)
        number = next(x)
        print(number)
    except StopIteration:
        break
```



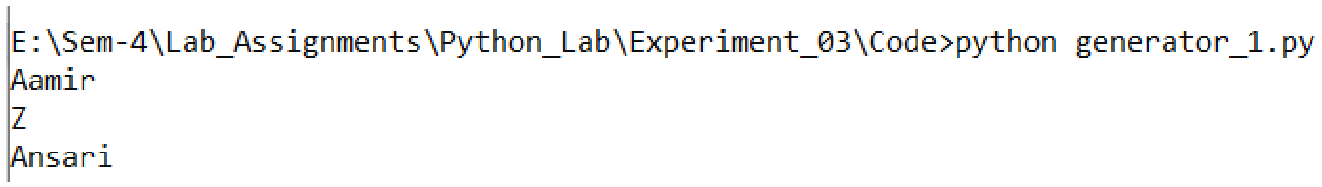
```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python iterator_2.py
Multiples of 5 are :
5
10
15
20
25
30
```

# Demonstration of generators (program 1)

```
def simpleGeneratorFun():
    yield "Aamir"
    yield "Z"
    yield "Ansari"

# x is a generator object
x = simpleGeneratorFun()

# Iterating over the generator object using next
print(x.__next__())
print(x.__next__())
print(x.__next__())
```



```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python generator_1.py
Aamir
Z
Ansari
```

# Demonstration of generators (program 2)

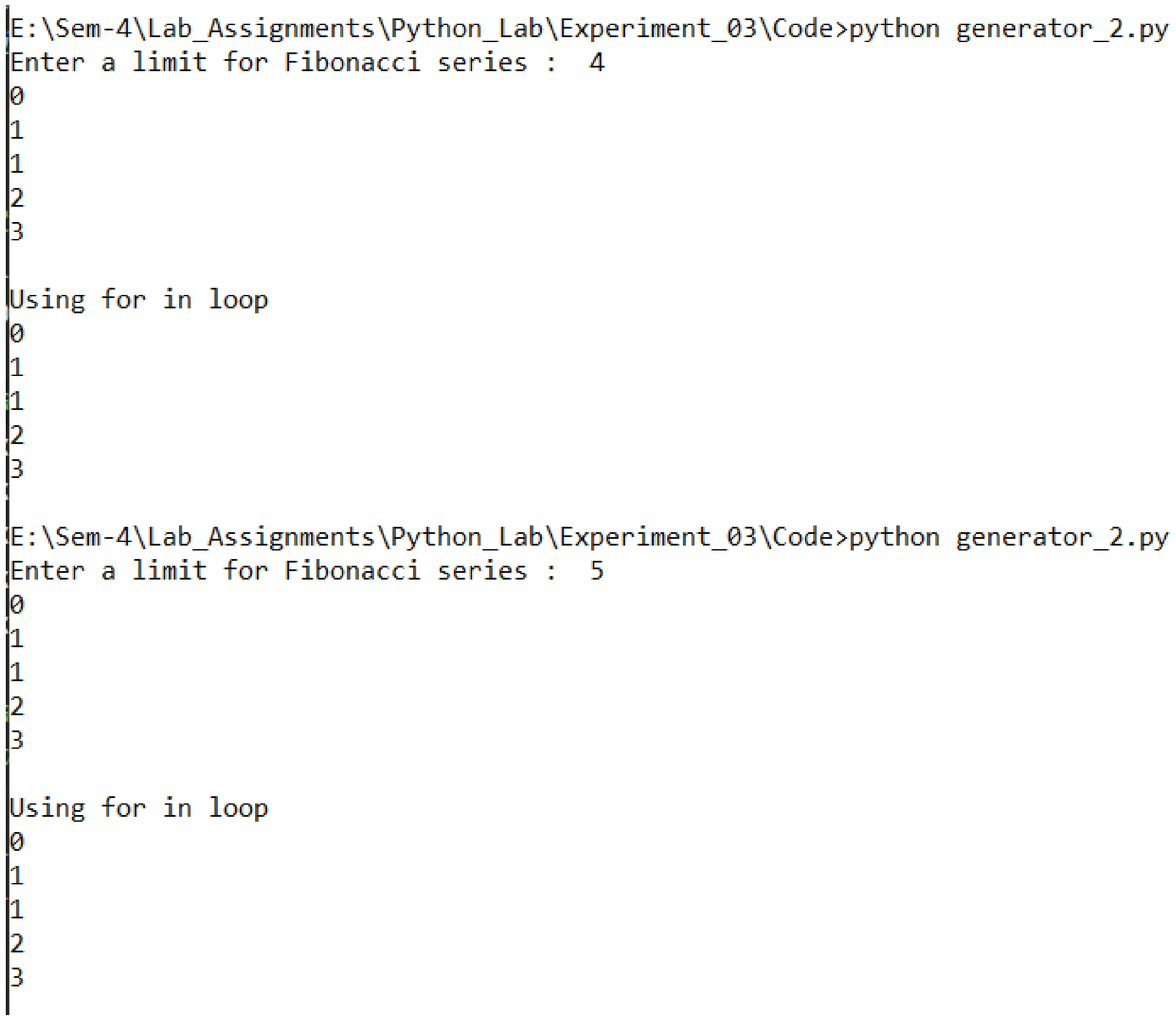
```
def fib(limit):
    # Initialize first two Fibonacci Numbers
    a, b = 0, 1

    # One by one yield next Fibonacci Number
    while a < limit:
        yield a
        a, b = b, a + b

# Create a generator object
lim = int(input("Enter a limit for Fibonacci series :  "))
x = fib(lim)

# Iterating over the generator object using next
print(x.__next__())
print(x.__next__())
print(x.__next__())
print(x.__next__())
print(x.__next__())

# Iterating over the generator object using for
# in loop.
print("\nUsing for in loop")
for i in fib(lim):
    print(i)
```



```
E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python generator_2.py
Enter a limit for Fibonacci series :  4
0
1
1
2
3

Using for in loop
0
1
1
2
3

E:\Sem-4\Lab_Assignments\Python_Lab\Experiment_03\Code>python generator_2.py
Enter a limit for Fibonacci series :  5
0
1
1
2
3

Using for in loop
0
1
1
2
3
```

<u>Conclusion</u>:

Hence we have successfully studied and used Decorators, Iterators and generators in python