Experiment 03

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

Roll No.	61
Name	V Krishnasubramaniam
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 atleast 2 python programs (for each) to understand the concept of Decorators, Iterators and Generators.

Introduction:

Decorators in Python

A decorator is a function that accepts a function as parameter and returns a function. A decorator takes the result of a function, modifies the result and returns it. Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

In Decorators, functions are taken as the argument into another function and then called inside the wrapper function.

Syntax:

```
@my_decorator
def hello_decorator():
   print("Decorator test")

#which is equivalent to the following code

def hello_decorator():
   print(" Decorator test")
hello_decorator = my_decorator(hello_decorator)
```

Example:

```
def hello_decorator(func):
   def inner1():
      print("Hello, this is before function execution")
      func()
      print("This is after function execution")
   return inner1
def function_to_be_used():
   print("This is inside the function!!")
function_to_be_used = hello_decorator(function_to_be_used)
function_to_be_used()
```

Output:

```
Hello, this is before function execution
This is inside the function!!
This is after function execution
```

Multiple decorators can be chained in Python. This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.

Example:

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner
def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner
@star
@percent
def printer(msg):
    print(msg)
printer("Hello")
```

Output:

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%******************************
```

Iterators in Python

Iterator in python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The iterator object is initialized using the iter() method. It uses the next() method for iteration. Lists, tuples, dictionaries, and sets 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.

The iterator object is initialized using the iter() method. It uses the next() method for iteration.

1. **__iter(iterable)__ method** that is called for the initialization of an iterator. This returns an iterator object
2. **next (__next__ in Python 3)** The next method returns the next value for the iterable. When we use a for loop to traverse any iterable object, internally it uses the iter() method to get an iterator object which further uses next() method to iterate over. This method raises a StopIteration to signal the end of the iteration.

The syntax of the iter() function is:

iter(object, sentinel)

Example:

```
iterable_value = 'Hello'
iterable_obj = iter(iterable_value)
while True:
   try:
      item = next(iterable_obj)
      print(item)
   except StopIteration:
      break
```

Output:

```
H
e
l
l
o
```

Building an iterator from scratch is easy in Python. We just have to implement the __iter__() and the __next__() methods.

Example:

```
class MyNumbers:
def __iter__(self):
self.a = 1
return self
def __next__(self):
x = self.a
self.a += 1
return x
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:

```
1
2
3
4
5
```

It is not necessary that the item in an iterator object has to be exhausted. There can be infinite iterators (which never ends). We must be careful when handling such iterators. The built-in function iter() can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.

Example:

```
int()                #Output: 0
inf = iter(int,1)
next(inf)            #Output: 0
next(inf)            #Output: 0
```

Generators in Python

Generators are functions that return a sequence of values. 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 yield, the function automatically 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.

Differences between Generator and Normal function:

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

Example:

```
# generator that returns characters from A to C.
def mygen():
yield 'A'
yield 'B'
yield 'C'
# call generator function and get generator object g
g = mygen()
# display all characters in the generator
print(next(g))  #Output: 'A'
print(next(g))  # display 'B'
print(next(g))  # display 'C'
```

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

Example:

```
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
```

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy. Similar to the lambda functions which create anonymous functions, generator expressions create anonymous generator functions. The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

Example:

```
my_list = [1, 3, 6, 10]
list_ = [x**2 for x in my_list]
generator = (x**2 for x in my_list)
print(list_)
print(next(generator))
print(next(generator))
print(next(generator))
print(next(generator))
```

Output:

```
[1, 9, 36, 100]
1
9
36
100
```

Results

Decorators:

1.

Program in decoratorExample1.py:

```
#to find execution time of a function
import time
import math

def calculate_time(func):
    def inner1(*args, **kwargs):
        begin = time.time()
        func(*args, **kwargs)
        end = time.time()
        print("Total time taken in : ", func.__name__, end - begin)
    return inner1

@calculate_time
def factorial(num):
    time.sleep(2)
    print("Factorial of",num,"=",math.factorial(num))

num = int(input("Enter a number:"))
factorial(num)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python decoratorExample1.py
Enter a number:10
Factorial of 10 = 3628800
Total time taken in :  factorial 2.014799118041992

C:\Users\vkris\Dropbox\Programming\Python\Experiments>python decoratorExample1.py
Enter a number:20
Factorial of 20 = 2432902008176640000
Total time taken in :  factorial 2.0146970748901367
```

2.

Program in decoratorExample2.py:

```
def hello_decorator(func):
    def inner1(*args, **kwargs):
        print("before Execution")
        returned_value = func(*args, **kwargs)
        print("after Execution")
        return returned_value
```

```
    return inner1

@hello_decorator
def sum_two_numbers(a, b):
    print("Inside the function")
    return a + b

a = int(input("Enter first number:"))
b = int(input("Enter second number:"))

# getting the value through return of the function
print("Sum =", sum_two_numbers(a, b))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python decoratorExample2.py
Enter first number:1
Enter second number:2
before Execution
Inside the function
after Execution
Sum = 3

C:\Users\vkris\Dropbox\Programming\Python\Experiments>python decoratorExample2.py
Enter first number:25
Enter second number:15
before Execution
Inside the function
after Execution
Sum = 40
```

3.

Program in decoratorExample3.py:

```
def hello_decorator(func):
    def inner1():
        print("Hello, this is before function execution")
        func()
        print("This is after function execution")
    return inner1
def function_to_be_used():
    print("This is inside the function!!")
function_to_be_used = hello_decorator(function_to_be_used)
function_to_be_used()
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python decoratorExample3.py
Hello, this is before function execution
This is inside the function!!
This is after function execution
```

4.

Program in decoratorExample4.py:

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner

@star
@percent
def printer(msg):
    print(msg)
printer("Hello")
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python decoratorExample4.py
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

Iterators:

1.

Program in iteratorExample1.py:

```
iterable_value = 'Hello'
iterable_obj = iter(iterable_value)
while True:
    try:
        item = next(iterable_obj)
```

```
        print(item)
    except StopIteration:
        break
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python iteratorExample1.py
H
e
l
l
o
```

2.

Program in iteratorExample2.py:

```
class MyNumbers:
def __iter__(self):
self.a = 1
return self
def __next__(self):
x = self.a
self.a += 1
return x
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python iteratorExample2.py
1
2
3
4
5
```

3.

Program in iteratorExample3.py:

```
class InfIter:
    def __iter__(self):
        self.num = 1
        return self
```

```
    def __next__(self):
        num = self.num
        self.num += 2
        return num
a = iter(InfIter())
print(next(a))
print(next(a))
print(next(a))
print(next(a))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python iteratorExample3.py
1
3
5
7
```

4.

Program in iteratorExample4.py:

```
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))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python iteratorExample4.py
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Generators:

1.

Program in generatorExample1.py:

```
def my_gen():
    n = 1
    print('This is printed first')
    yield n
    n += 1
    print('This is printed second')
    yield n
    n += 1
    print('This is printed at last')
    yield n

for item in my_gen():
    print(item)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python generatorExample1.py
This is printed first
1
This is printed second
2
This is printed at last
3
```

2.

Program in generatorExample2.py:

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

for char in rev_str("hello"):
    print(char)
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python generatorExample2.py
o
l
l
e
h
```

3.

Program in generatorExample3.py:

```
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:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python generatorExample3.py
4895
```

4.

Program in generatorExample4.py:

```
def numberGenerator(n):
  try:
     number = 0
     while number < n:
         yield number
         number += 1
  finally:
     yield n

print(list(numberGenerator(10)))
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python generatorExample4.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

5.

Program in generatorExample5.py:

```
def fibo():
    first,second=0,1
    while True:
        yield first
        first,second=second,first+second

for x in fibo():
    if x>50:
        break
    print(x, end=" ")
```

Output:

```
C:\Users\vkris\Dropbox\Programming\Python\Experiments>python generatorExample5.py
0 1 1 2 3 5 8 13 21 34
```

Conclusion:

Thus, we have understood the concepts of decorators, iterators and generators in Python, along with their practical uses by performing hands-on programs.