16-5-23

Decorator:

-> Decorator is the function used to give the additional set of properties/features/functionalities to the other function/class.

-> Decorator is a function which can take a function as arguments and extends its functionality and returns modified function with extended functionality.

-> The main objective of decorator functions is that we can extend the functionality of existing functions or classes without modifying the main function functionality/class functionality.

-> There are two types of decorators:

1. Built-in decorator
2. User-defined decorator

Built-in decorator: @classmethod,
@staticmethod, @abstractmethod,
@property method.

@ ---> decorator is ~~defined~~ identified with @ symbol.

user-defined decorator ~~function~~ written with the function syntax:

```
def deco-name(func):
    def inner_method(arguments):
        before task
        func(arguments)
        after task
    return inner_method
```

user-defined decorator to a class are written with the function syntax:

```
def deco-name(cls):
    def inner_method(arguments):
        before task
        obj = cls(arguments)
        after task
```

return obj
return inner_method.

Decorator to a Class:

It is a method used to give the additional set of features or functionality to a class which is like a decorator to a function here. It carries the address of the class to the decorator function/method needed to decorate the class. The inner function consists of the code or the instructions that need to be executed before object creation. The object creation will happen inside the inner method itself.

Syntax:

```
def deco_name(cls):
    statements
    def inner_method(*args, **kwargs):
        Before the object creation.
        obj = cls(*args, **kwargs)
        # object creation happend
```

after object creation
return obj

return inner_method

@deco_name
class Cname:
    |
    |
    |

Cname = deco_name (cname)

obj = Cname(arguments)

Syntax :-

①

```
def outer(func/cls):
    def inner(*args, **kwargs)
        print("before execution")
        func(*args, **kwargs)
        print("after execution")
    return inner
outer(existing func address)
```

②

```
def outer(func/cls):
    def inner(*args, **kwargs)
        print("before execution")
        res = func(*args, **kwargs)
```

```
        print ("after execution")
        return res
    return inner
outer(existing func address)
```

example:

①

```
def outer(func):
    def inner(*args, **kwargs):
        print ("hai")
        print (func)
        print (func(*args, **kwargs))
        print ("bye")
    return inner

@outer
def sample(a,b):
    return a+b
sample(10,20)
```

②

```
def outer(func):
    def inner(*args, **kwargs):
        print ("hai")
        print (func)
        res = func(*args, **kwargs)
```

```
        print("bye")
        return res
    return inner

@outer
def sample(a, b):
    return a+b

print(sample(10, 20))
```

O/P :-

hai

<function sample at 0x00>

bye

30

17-5-23.

```
from datetime import datetime
def timer(func):
    def inner(*args, **kwargs):
        a = datetime.now()
        res = func(*args, **kwargs)
        b = datetime.now()
        print(a, b, b-a)
        return res
    return inner

@timer
def demo(a, b):
    l = [ ]
```

```
    for i in range(a,b):
        l+=[i]
    return l
demo(1,100000)
```

o/p :-

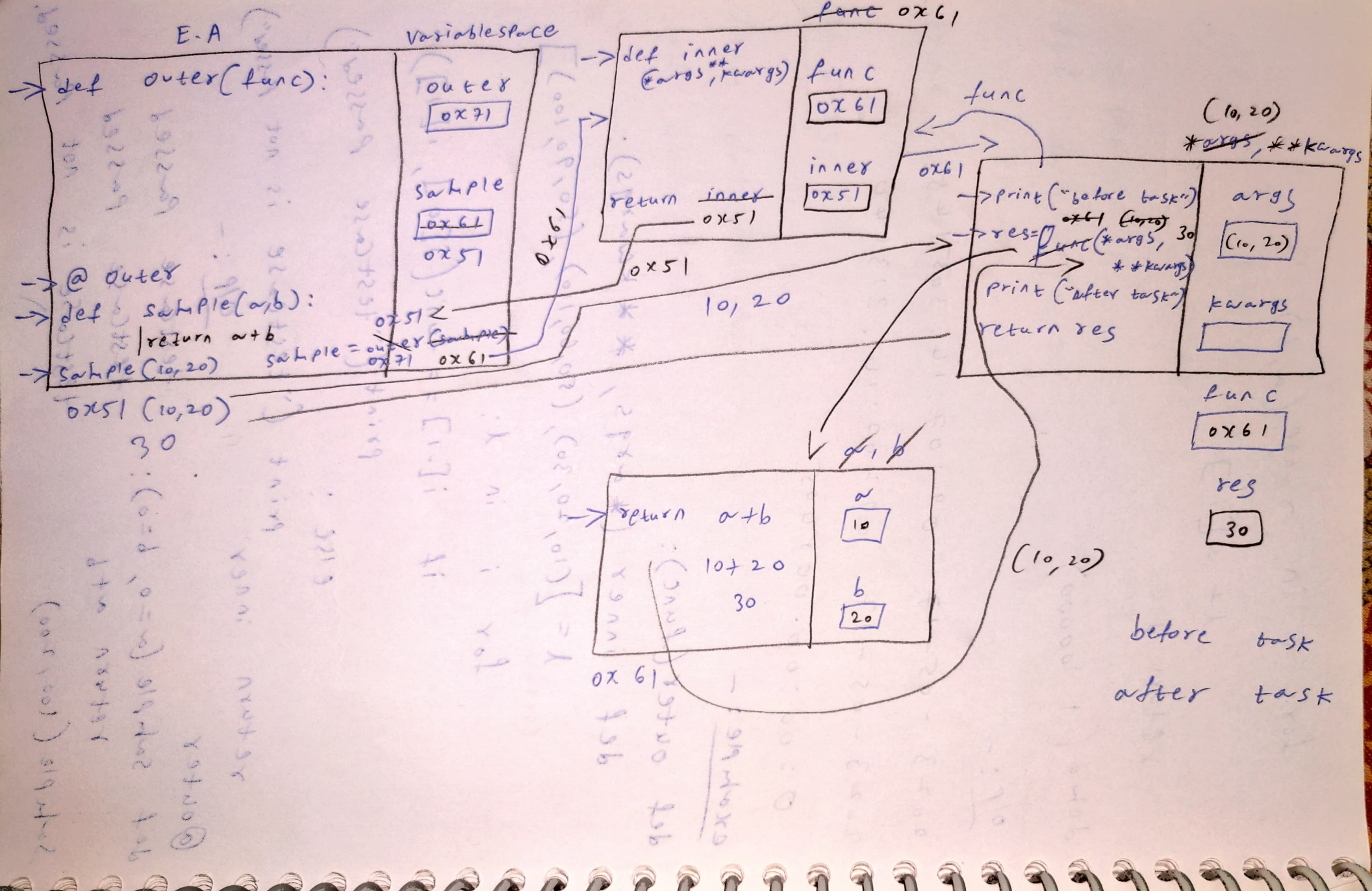
```
2023-05-17 09:07:14.305145
2023-05-17 09:07:14.313640
0:00:00.008495
```

example :-

```
def outer(func):
    def inner(*args, **kwargs):
        l=[(10,20,30),(50,60,110),(50,90,100)]
        for i in l:
            if i[-1]==func(i[0],i[1]):
                print("testcase passed")
            else:
                print("testcase is not passed")
    return inner
@outer
def sample(a=0,b=0):
    return a+b
sample(100,200)
```

o/p :-

```
testcase passed
testcase passed
testcase is not passed.
```


E.A
Variablespace
def outer(func):
outer
0x71
sample
0x61
0x51
@ outer
def sample(a,b):
return a+b
sample(10,20)
sample = outer(sample)
0x71 0x61
0x51 (10,20)
30
def inner (*args,**kwargs)
return inner
0x51
func
0x61
inner
0x51
0x61
0x51
10, 20
func
0x61
(10,20)
*args, **kwargs
print("before task")
res=func(*args, **kwargs)
print("after task")
return res
args
(10,20)
kwargs
func
0x61
res
30
(10,20)
a, b
return a+b
10+20
30
a
10
b
20
0x61
before task
after task

decorator memory management

14-2-23

Generators :

-> Python provides a generator to create your own generator function.

-> Generators are the functions that are used to create a sequence by giving the value one after the other.

-> A generator is a special type of function which does not return a single value, instead, it returns a generator object (generator) with a sequence of values.

-> If we wanted to create a generator inside the function we must use the keyword called yield.

-> Yield is the predefined keyword. It is traversing the value from one location to another location.

-> It is used to pause and get the control back to the current execution location to store the user call comes and the function control again does back and executes that the position where it has stopped.

19-5-23.

Generators :

-> Python provides a generator to create your own generator function.

-> Generators are the functions that are used to create a sequence by giving the value one after the other.

-> A generator is a special type of function which does not return a single value, instead, it returns a generator object (generator) with a sequence of values.

-> if we wanted to create a generator, inside the function we must use the keyword called yield.

-> Yield is the predefined keyword it is traversing the value from one location to another location.

-> It is used to pause and get the control from the current execution location to where the user calls comes and the function control again goes back and executes from the position where it has stopped.

-> yield keyword iterating the value one space to another space.

-> once after the generator's execution we need to give that generated object to the static iterator or dynamic iterator or type casting for traversing.

-> The generator object we can utilize only once.

-> In a generator function, a yield statement is used rather than a return statement.

Example:

```
def gen_demo( ): # is called as generator function
    print ("first pause")
    yield
    print ("second pause")
    yield
    print ("third pause")
    yield
    print ("termination")
```

```
oa = gen_demo()
print(oa)
i = iter(oa)
print(next(i))
print(next(i))
```

output:

<generator object gen_demo at 0x7fbbf0...>

first Pause

None

Second Pause

None.

examples:

①

```
def Sample(a,b):
    for i in range(a,b):
        yield i

Sample(5,10)
```

O/P :-

<generator object Sample at 0x....>

②

```
def Sample(a,b):
    yield a
    print('a',a)
    yield b
    print('b',b)
for i in Sample(5,10):
    print(i)
```

o/p :-

```
5
a 5
10
b 10
```

③

```
def sample(a,b):
    yield a
    print('a',a)
    yield b
    print('b',b)
list(sample(5,10))
```

o/p :-

```
a 5
b 10
[5,10]
```

④

```
def sample(a,b):
    yield a
    print('a',a)
    yield b
    print('b',b)
res = sample(5,10)
print(list(res))
print(list(res))
```

o/p :-

```
a 5
b 10
[5,10]
[]
```

```
def sample(a,b):
    for i in range(a,b):
        return i
```

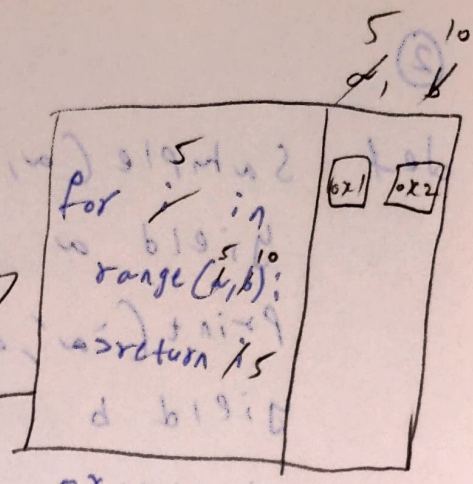
sample(5,10)

0x171(5,10)

5

5,10

5



generator :

```
def sample(a,b):
    for i in range(a,b):
        yield i
res = sample(5,8)
```

0x191(5,8)

sample

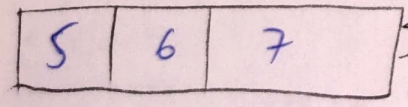
0x191

5,8

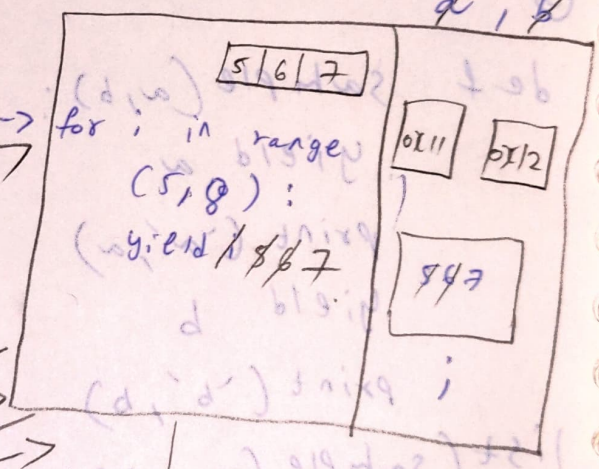
5

6

7



generator object



None.

20-5-23.

Program :

```
def bottle ( ):
    l = [100, 'HAI', 150, 160.0]
    for i in l :
        if type(i) == int :
            yield i

def sample(val):
    return val + val

for i in bottle ( ):
    print(sample(i))
```

o/p :-

200
300.

Program :-

```
l = ['apple', 'bat', 'cat', 'egg', 'icecream',
    'veg', 'fishcurry']

def demo (coll):
    for i in coll:
        if len(i) % 2 == 0 and i[0] in 'aeiou':
            yield i

demo (l)
print(list(demo(l))
```

o/p :-

['icecream']

```
l = ['apple', 'bat', 'Cat', 'egg', 'icecream',
     'veg', 'fishcurry']
st = (i for i in l if len(i)%2==0
      and i[0] in 'AEIOUaeiou')
print(st)
```

o/p :-

<generator object .....>

```
st = [i for i in l if len(i)%2==0
      and i[0] in 'AEIOUaeiou']
print(st)
```

o/p :-

['icecream']

```
res = map(lambda var:var if len(var)%2
      ==0 and var[0] in 'aeiouAEIOU'
      else None, l)
print(list(res))
```

o/p :-

[None, None, None, None, 'icecream',
None, None]

```
res = filter(lambda var:var if len(var)%
      2==0 and var[0] in
      'aeiouAEIOU' else None, l)
print(list(res))
```

o/p :-

['icecream']

```
l = ['apple' - . . . . . ]

res = map (lambda var : var.title() if
        len(var) %. 2 == 0 and var[0]
        in 'aeiouAEIOU' else None, l)

temp = filter (lambda var : var, list(res))

print (list(temp)).
```

o/p :-

['Icecream']

Ternary operator :-

[None, "haii"] [len("ab") %. 2 == 0 and
'ab'[0] in 'aeiouAEIOU']

```
>>> "haii"

l = [1, 2]                    l [False]

l[True]                       >>> 1

>>> 2

l = [1,2]          [1,2][0]          [1,2][1]

l [1]

>>> 1                    >>> 2

>>> 2
```

-> It is a conditional operator based on the condition it returns the value.

-> In ternary operator we are using collection with indexing.

-> In ternary operator the combination of collection datatypes with operators with indexing concept.

Syntax :-

Collection Indexing operator

[False stat, true stat] [<condition/expression>]

{True : True Stat, False : False Stat.} [<condition/expression>]

(False Stat, True Stat) [<condition/expression>].

22-5-23.

Iterator :-

-> Iteration: It is a process of traversing through the sequence to get the value one after another.

-> Iterator: the function that helps to perform the iteration are called iterator.

-> Iterator is a function that is used to traverse through the collection/sequence

and get the value one after another,

-> these are the functions who are working behind the for looping statement and providing the values to the loop one by one.

-> iterators are two types

① Static iterator

② dynamic iterator

-> For loop mainly works on the concept of iterator in dynamic nature.

-> In iterator there are two types of functions which plays the major role :-

Ⓐ. iter (collection):

-> It is a predefined function.

-> It helps to initialize the cursor to the beginning of the sequence and returns the sequence of value reference address.

Syntax :-

```
var = iter(collection)
```

Note : collection/multi-value/iterable.

-> It is an inbuilt function which will create an object and store the address of the iterative sequence inside the iterator object.

(B). next (iterator object) :-

-> It is a predefined function. It helps us to traverse the sequence by giving the element / value one after another.

Syntax :

next(var)

Note : var will be the address which we have got from the iter function. this helps you to move the cursor element by element.

when it is moving from element to element it gives the value of the current position.

-> By using this next() we travel / traverse through the sequence / collection.

-> It will reach the address present in the object.

-> It will fetch the data from the memory address block

-> It will send points to the next node.

Iterator object :-

-> The address returned by the iter()

function needs to be stored in a variable and that variable is called an iterator object.

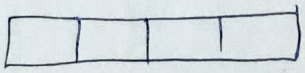
-> Iterables :- The variables which can undergo the process of iteration are called iterables.

Note :- input for the iter() function must be a collection.

-> In python if a class contains __iter__ and __next__ (next).

dynamic iterator :

for i in coll:



var iter() <- it will accept only iterable value

it holds the reference address <- it will creates a raw cursor

next(i) it will assign to start of the collection

-> it will take the reference address from iterable object.

-> it will fetch and return the value from value space by the help of address. <- it return current value with reference address.

-> It will change the cursor from current position to next position.

-> It will move only one step.

Static iterator :

iterable object

var = iter(collection)

iterator function

any collection datatype.

next(var)

↓

function

↓

fetch / return / change cursor.

Example :-

```
var = iter('hello')
print(var)
print(next(var))
print(next(var))
print(next(var))
print(next(var))
print(next(var))
print(next(var))
```

O/P :-

<str_ascii_ ..... >

h
e
l
l
o

Stop Iteration :

23-5-23.

①

```
for i in {'a':10, 'b':20, 'c':30, 'd':40}:
    print(i)

res = iter({'a':10, 'b':20, 'c':30, 'd':40})
l = [next(res), next(res), next(res), next(res)]
print(l)
```

o/p :-

a
b
c
d

['a', 'b', 'c', 'd']

②

```
d = {'a':10, 'b':20, 'c':30, 'd':40}
res = iter(d)
print(d[next(res)])
print(d[next(res)])
print(d[next(res)])
print(d[next(res)])
```

```
res = iter({'a':10, 'b':20, 'c':30,
            'd':40}.values())
print(next(res))
print(next(res))
print(next(res))
print(next(res))
```

O/P :-

10
20
30
40
10
20
30
40.

Program :- Custom iterators

User defined range function :-

```
class series:
    def __init__(self, start=0, end=20, up=1):
        self.start = start
        self.stop = end
        self.up = up
    def __iter__(self):
        return self
    def __next__(self):
        if self.start <= self.stop:
            val = self.start
            self.start += self.up
            return val
```

```
        else:
            raise    StopIteration.

for i in series(0, 10, 2):
    print(i)

#res = iter(series())
# Print(next(res))
# Print(next(res))
```

iterator Memory management.