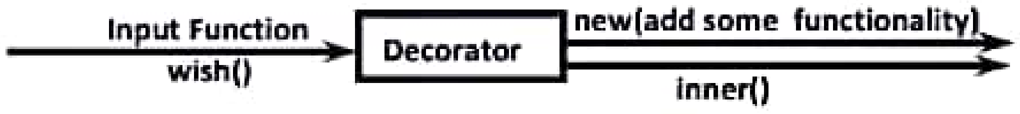
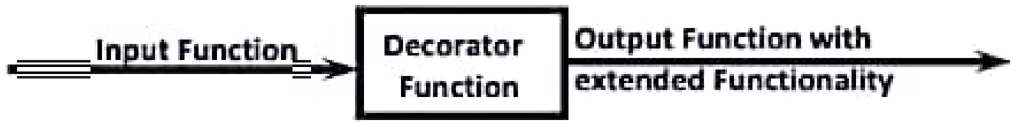
DECORATOR FUNCTIONS

Decorator is a function which can take a function as argument and extend its functionality and returns modified function with extended functionality.





The main objective of decorator functions is we can extend the functionality of existing functions without modifies that function.

```
def wish(name):
  print("Hello",name,"Good Morning")
```

This function can always print same output for any name

Hello Durga Good Morning
Hello Ravi Good Morning
Hello Sunny Good Morning

But we want to modify this function to provide different message if name is Sunny.
We can do this without touching wish() function by using decorator.

```
def decor(func):
  def inner(name):
    if name=="Sunny":
      print("Hello Sunny Bad Morning")
    else:
      func(name)
  return inner

@decor
def wish(name):
  print("Hello",name,"Good Morning")

wish("Durga")
wish("Ravi")
wish("Sunny")
```

Output
Hello Durga Good Morning
Hello Ravi Good Morning

Hello Sunny Bad Morning
In the above program whenever we call wish() function automatically decor function will be executed.

How to call Same Function with Decorator and without Decorator:

We should not use @decor

```
def decor(func):
  def inner(name):
    if name=="Sunny":
      print("Hello Sunny Bad Morning")
    else:
      func(name)
  return inner

def wish(name):
  print("Hello",name,"Good Morning")

decorfunction=decor(wish)

wish("Durga") #decorator wont be executed
wish("Sunny") #decorator wont be executed

decorfunction("Durga")#decorator will be executed
decorfunction("Sunny")#decorator will be executed
```

Output

```
Hello Durga Good Morning
Hello Sunny Good Morning
Hello Durga Good Morning
Hello Sunny Bad Morning
```

```
def  smart_division(func):
  def inner(a,b):
    print("We are dividing",a,"with",b)
    if b==0:
      print("OOPS...cannot divide")
      return
    else:
      return func(a,b)
  return inner

@smart_division
def  division(a,b):
  return a/b
```

```
print(division(20,2))
print(division(20,0))
```

Without Decorator we will get Error. In this Case Output is:

```
10.0
Traceback (most recent call last):
File "test.py", line 16, in <module>
  print(division(20,0))
File "test.py", line 13, in division
  return a/b
ZeroDivisionError: division by zero
```

With Decorator we won't get any Error. In this Case Output is:

```
We are dividing 20 with 2
10.0
We are dividing 20 with 0
OOPS...cannot divide
None
```

```python
def marriagedecor(func):
    def inner():
        print('Hair decoration...')
        print('Face decoration with Platinum package')
        print('Fair and Lovely etc..')
        func()
    return inner

def getready():
    print('Ready for the marriage')

decorated_getready=marriagedecor(getready)

decorated_getready()
```

Decorator Chaining

We can define multiple decorators for the same function and all these decorators will form Decorator Chaining.

Eg:

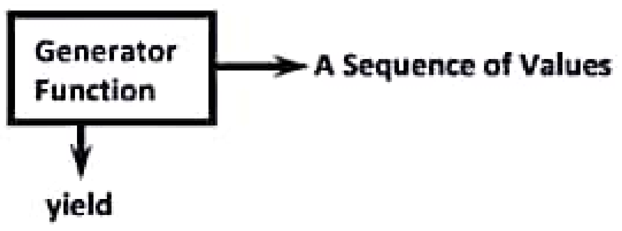
```python
@decor1
@decor
def num():
```

For num() function we are applying 2 decorator functions. First inner decorator will work and then outer decorator.

```
def decor1(func):
   def inner():
      x=func()
      return x*x
   return inner

def decor(func):
   def inner():
      x=func()
      return 2*x
   return inner

@decor1
@decor
def num():
   return 10

print(num())
```

GENERATOR FUNCTIONS

Generator is a function which is responsible to generate a sequence of values.
We can write generator functions just like ordinary functions, but it uses yield keyword to return values.



```python
def mygen():
    yield 'A'
    yield 'B'
    yield 'C'

g=mygen()
print(type(g))

print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

Output

```
<class 'generator'>
A
B
C
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print(next(g))
StopIteration
```

```python
def countdown(num):
    print("Start Countdown")
    while(num>0):
        yield num
        num=num-1
values=countdown(5)
for x in values:
    print(x)
```

Output

```
Start Countdown
5
```

```
4
3
2
1
```

Eg 3: To generate first n numbers

```python
def firstn(num):
    n=1
    while n<=num:
        yield n
        n=n+1

values=firstn(5)
for x in values:
    print(x)
```

Output

```
1
2
3
4
5
```

We can convert generator into list as follows:

```python
values = firstn(10)
l1 = list(values)
print(l1)   #[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eg 4: To generate Fibonacci Numbers...
The next is the sum of previous 2 numbers

Eg: 0,1,1,2,3,5,8,13,21,...

```python
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)
```

Output

```
0
1
1
2
3
5
8
13
21
34
55
89
```

Advantages of Generator Functions:

1) When compared with Class Level Iterators, Generators are very easy to use.
2) Improves Memory Utilization and Performance.
3) Generators are best suitable for reading Data from Large Number of Large Files.
4) Generators work great for web scraping and crawling.

Generators vs Normal Collections wrt Performance:

```python
import random
import time

names = ['Sunny','Bunny','Chinny','Vinny']
subjects = ['Python','Java','Blockchain']

def people_list(num_people):
    results = []
    for i in range(num_people):
        person = {
                    'id':i,
                    'name': random.choice(names),
                    'subject':random.choice(subjects)
                  }
        results.append(person)
    return results

def people_generator(num_people):
    for i in range(num_people):
        person = {
                    'id':i,
                    'name': random.choice(names),
                'major':random.choice(subjects)
              }
        yield person

"""t1 = time.clock()
people = people_list(10000000)
t2 = time.clock()'''

t1 = time.clock()
people = people_generator(10000000)
t2 = time.clock()

print('Took {}'.format(t2-t1))
```

Note: In the above program observe the differnce wrt execution time by using list and generators

Generators vs Normal Collections wrt Memory Utilization:

Normal Collection:

```
l=[x*x for x in range(10000000000000000)]
print(l[0])
```

We will get MemoryError in this case because all these values are required to store in the memory.

Generators:

```
g=(x*x for x in range(10000000000000000))
print(next(g))
```

Output: 0

We won't get any MemoryError because the values won't be stored at the beginning