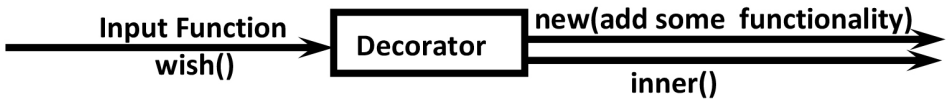
Function Decorators:

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.

```
1)  def smart_division(func):
2)      def inner(a,b):
3)          print("We are dividing",a,"with",b)
4)          if b==0:
5)              print("OOPS...cannot divide")
6)              return
7)          else:
8)              return func(a,b)
9)      return inner
10)
11) @smart_division
12) def division(a,b):
13)     return a/b
14)
15) print(division(20,2))
16) print(division(20,0))
17)
18) without decorator we will get Error.In this case output is:
19)
20) 10.0
21) Traceback (most recent call last):
22)   File "test.py", line 16, in <module>
23)     print(division(20,0))
24)   File "test.py", line 13, in division
25)     return a/b
26) 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

Decorator Chaining

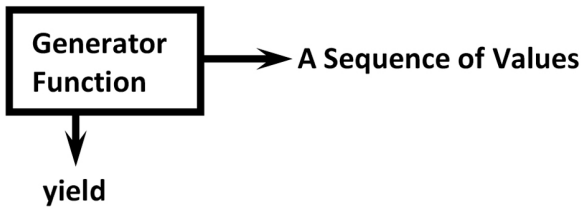
We can define multiple decorators for the same function and all these decorators will form Decorator Chaining.

Eg:
@decor1
@decor
def num():

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

Generators

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.



To generate first n numbers:

```
1)  def firstn(num):
2)      n=1
3)      while n<=num:
4)          yield n
5)          n=n+1
6)
7)  values=firstn(5)
8)  for x in values:
9)      print(x)
10)
11) Output
12) 1
13) 2
14) 3
15) 4
16) 5
```

We can convert generator into list as follows:

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