DECORATORS, ITERATORS, GENERATORS

1. DECORATOR

⇨ A decorator is a function that adds extra features to the existing function without changing the original code

⇨ Syntax:

```
@decorator_name
def function_name():
        #code
```

Ex:

```
def my_decorator(func):
    def wrapper():
        print("party starts")
        func()
        print("party ends+")
    return wrapper
@my_decorator
def say_hlo():
    print("hello everyone")
say_hlo()
```

2. ITERATOR

⇨ An iterator is used to iterate through the sequence but one at a time

⇨ We create the iterator by using iter() function

⇨ iter() function is the built in function

⇨ syntax:

```
iter(sequence)
```

- this function returns the iterator object
- this iterator is used to iterate over the each element in a sequence

⇨ we can fetch the values one at a time by using “next() function or __next__()”

⇨ syntax:

```
next(iterator_object) or iterator_object.__next__()
```

- this next() and __next__() holds the previous value
- when you call the next() it gives the next item in a sequence
- it stops when there are no items and it raises an “StopIteration” error

ex:

```
nums=[1,2,3,4,5,6,7]
iterator_obj=iter(nums)
print(iterator_obj.__next__())
print(iterator_obj.__next__())
print(next(iterator))
print(next(iterator))
```

⇨ we can also create our own iterator

⇨ creating a custom iterator with class

ex:

```
class Topten:
    def __init__(self):
        self.num=1
    def __iter__(self):
        return self
    def __next__(self):
        if self.num<=10:
            val = self.num
            self.num+=1
            return val
        else:
            raise StopIteration
values=Topten()
for i in values:
    print(i)
```

3. GENERATOR

⇨ A generator is a special kind of function that gives/generates the values one by one when you call/use it

⇨ It is like an iterator

⇨ Syntax:

```
def function_name():
        yield value1
        yield value2
```

- yield is like return, but it pauses the function and remembers where it left off
- yield will return a value and pauses the function
- when you call it , it resumes the process from where it was stopped
- difference b/w the return and yield

return	**yeild**
it ends the function	It pauses the function
It returns one value	It returns multiple values but one by one
It can’t resume	Can resume from where it stopped

Ex:

```
def my_generator():
        for i in range(1,11):
            yield i
generator_obj=my_generator()
print(generator_obj.__next__())
print(generator_obj.__next__())
print(next(generator_obj
```