What is a Generator

Python generators are a simple way of creating iterators.

```
# iterable
class mera_range:
    def __init__ (self,start,end):
        self.start=start
        self.end=end

    def __iter__(self):
        return mera_range_iterator(self)

#iterator
class mera_range_iterator:
    def __init__(self,iterable_obj):
        self.iterable=iterable_obj

    def __iter__(self):
        return self

    def __next__(self):

        if self.iterable.start >= self.iterable.end:
            raise StopIteration

        current=self.iterable.start
        self.iterable.start+=1
        return current
```

A simple Example of Generator

```
def gen_demo():

    yield "First statement"
    yield "second statement"
    yield "third statement"



# normal function me return hota hai but generator me yield hota hai
and there can be multiple yield statement

gen=gen_demo()
print(gen)
```

```
<generator object gen_demo at 0x0000025440121A60>
```

```
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
First statement
second statement
third statement
```

```
---------------------------------------------------------------------------
-----
StopIteration                             Traceback (most recent call
last)
Cell In[34], line 4
      2 print(next(gen))
      3 print(next(gen))
----> 4 print(next(gen))

StopIteration:
```

```
# jab aap call karte ho apni generator ko to wo palat ke ek generator
object deta hai and aap uss generator object ke upar next
# next laga ke item ko print kar sakte ho

gen=gen_demo()
for i in gen:
    print(i)
```

```
First statement
second statement
third statement
```

python tutor Demo (yield vs return)

Example 2

```
def square(num):
    for i in range(1,num+1):
        yield i**2

gen=square(10)

print(next(gen))
print(next(gen))
print(next(gen))


for i in  gen:
    print(i)
```

```
# aap dekhoge ki jab loop suru hua to wo 1 se suru nahi hua hai balki
print statement jo 3 tak ka square print kara dya hai uske baad se
suru hua hai
```

```
1
4
9
16
25
36
49
64
81
100
```

Range Function using Generator

```
def mera_range(start,end):
    for i in range(start,end):
        yield i

for i in mera_range(15,26):
    print(i)
```

```
15
16
17
18
19
20
21
22
23
24
25
```

Generator Expression

```
# list comprehension
L=[i**2 for i in range (1,101)]
L
```

```
[1,
 4,
 9,
 16,
 25,
 36,
 49,
 64,
```

```
81,
100,
121,
144,
169,
196,
225,
256,
289,
324,
361,
400,
441,
484,
529,
576,
625,
676,
729,
784,
841,
900,
961,
1024,
1089,
1156,
1225,
1296,
1369,
1444,
1521,
1600,
1681,
1764,
1849,
1936,
2025,
2116,
2209,
2304,
2401,
2500,
2601,
2704,
2809,
2916,
3025,
3136,
3249,
```

```
 3364,
 3481,
 3600,
 3721,
 3844,
 3969,
 4096,
 4225,
 4356,
 4489,
 4624,
 4761,
 4900,
 5041,
 5184,
 5329,
 5476,
 5625,
 5776,
 5929,
 6084,
 6241,
 6400,
 6561,
 6724,
 6889,
 7056,
 7225,
 7396,
 7569,
 7744,
 7921,
 8100,
 8281,
 8464,
 8649,
 8836,
 9025,
 9216,
 9409,
 9604,
 9801,
 10000]

gen=(i**2 for i in range(1,101))

for i in gen:
    print(i)
```

```
1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
400
441
484
529
576
625
676
729
784
841
900
961
1024
1089
1156
1225
1296
1369
1444
1521
1600
1681
1764
1849
1936
2025
2116
2209
2304
2401
2500
```

```
2601
2704
2809
2916
3025
3136
3249
3364
3481
3600
3721
3844
3969
4096
4225
4356
4489
4624
4761
4900
5041
5184
5329
5476
5625
5776
5929
6084
6241
6400
6561
6724
6889
7056
7225
7396
7569
7744
7921
8100
8281
8464
8649
8836
9025
9216
9409
9604
```

```
9801
10000

(i**2 for i in range(1,101))

<generator object <genexpr> at 0x0000025440130EE0>
```

Practical Example

```
## suppose aapko koi model train  karna hai and usme bahut sare pic
hai
#to one by one hum photo ko load karke uspe koi bhi operation perform
kar sakte hain
```

Benefits of using a Generator

1. Ease of Implementation

2. Memory Efficient

```
L = [x for x in range(100000)]
gen = (x for x in range(100000))

import sys

print('Size of L in memory',sys.getsizeof(L))
print('Size of gen in memory',sys.getsizeof(gen))

Size of L in memory 800984
Size of gen in memory 192
```

3. Representing Infinite Streams

```
def all_even():
    n = 0
    while True:
        yield n
        n += 2

even_num_gen = all_even()
next(even_num_gen)
next(even_num_gen)

2
```

4. Chaining Generators

```
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))))
```

```
4895
```