```
generators is a simple way of creating iterators
```

In [ ]:

why Iterator & generator

In [1]:
```
l1 = [i for i in range(1,5000)]
for i in l1:
    i**2
```

In [2]:
```
import sys
sys.getsizeof(l1)
```

Out[2]: 43032

In [3]:
```
l2 = range(1,5000)
for i in l2:
    i**2
```

In [4]:
```
# using iterator the memory consumption is min
# because only one item stays in memory at a time
# then removes it and move another item to memory

import sys
sys.getsizeof(l2)
```

Out[4]: 48

In [ ]:

Creating a Range Function using iterators

In [2]:
```
# Iterable
class MeraRange:
    def __init__(self,start,end):
        self.start = start
        self.end = end

    def __iter__(self):
        return MeraIterator(self)
```

In [3]:
```
# Iterator
class MeraIterator:
    def __init__(self,iterator_obj):
        self.iterator = iterator_obj

    def __iter__(self):
        return self

    def __next__(self):
        if self.iterator.start>=self.iterator.end:
            raise StopIteration

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

In [4]:
```
MeraRange(1,10)
```

Out[4]:
```
<__main__.MeraRange at 0x7f0ee40dbd30>
```

In [6]:
```
for i in MeraRange(1,10):
    print(i,end=' ')
```

```
1 2 3 4 5 6 7 8 9
```

In [ ]:
```

```

Creating a Function using generators

In [8]:
```
# creating a generator fuctn

def gen_demo():
    yield 'first line'
    yield 'second line'
    yield 'third line'
```

In [11]:
```
gen = gen_demo()
gen
```

Out[11]:
```
<generator object gen_demo at 0x7f0ee51aee40>
```

In [14]:
```
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call l
ast)
Input In [14], in <module>
----> 1 print(next(gen))
      2 print(next(gen))
      3 print(next(gen))

StopIteration:
```

or.

In [15]:
```
gen = gen_demo()
gen
```

Out[15]:
```
<generator object gen_demo at 0x7f0ec664df20>
```

In [16]:
```
for i in gen:
    print(i)
```

```
first line
second line
third line
```

In [ ]:

Old revision: Creating an iterator

In [13]:
```
no = [1,2,3,4,5,6]

# fetch the iterator
iter_no = iter(no)

print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
print(next(iter_no))
```

```
1
2
3
4
5
6

---------------------------------------------------------------------------
----
StopIteration                             Traceback (most recent call l
ast)
Input In [13], in <module>
     10 print(next(iter_no))
     11 print(next(iter_no))
---> 12 print(next(iter_no))

StopIteration:
```

In [ ]:

```
Note :

- Generator is a fucnt that does not have a return statement insted of
that it has a return statement
```

In [ ]:

```
Differnce between Generator & normal fuctn
```

- Noraml fuctn ek baar kaam kr ke memory sai nikla jata hai where as the yeild hota hai wo partially apna kaam karta hia phir memory sai nikalta hai but apna purana state aur variable ki value yaad rakhta hai. jis sai jab wo wapas generator fucntn mai jata hai toh jaha sai uska kaam baki tha wo continue karne lagta hai

In [ ]:

In [17]:
```
# create a fibronaaci code
# List of fib numbers up to 500: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

no = int(input('generate fibonacci no. upto : '))
f = 0
s = 1

for i in range(no):
    if f<no:
        print(f, end =' ')
        tmp = f
        f = s
        s = tmp + s
```

```
generate fibonacci no. upto : 500
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

In [ ]:

In [ ]:

Just as It is pal

program to generate fibonacci series

In [20]:
```
# Generate no. of fibonaaci no's

no = int(input('Enter the no : '))
f = 0
s = 1
for i in range(no):
    print(f, end = ' ')
    tmp = f
    f = s
    s = s+tmp
```

```
Enter the no : 5
0 1 1 2 3
```

Generator fuctn for fibo-nacci series

In [34]:
```
def genfibo(no):
    f = 0
    s = 1
    while f<no:
        yield f
        tmp = f
        f = s
        s = tmp+s
```

In [35]:
```
x = genfibo(5)  # here the generator obj is created and get stored on x

# calling generator using next method
print(next(x)) # here the fuctn gets called only when next is used
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
0
1
1
2
3
```

or.

In [ ]:

In [ ]:

Creating a custom range fucntn using generator

Ex 1

In [21]:
```
def meraRangefuctn(no):
    i = 0
    while i<no:
        yield i
        i+=1
    return # Raise stopIteration
```

In [24]:
```
for i in meraRangefuctn(5):
    print(i, end=' ')
```

```
0 1 2 3 4
```

or

In [28]:
```
x = meraRangefuctn(4)

# calling generator using next method
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
0
1
2
3

---------------------------------------------------------------------------
----
StopIteration                             Traceback (most recent call l
ast)
Input In [28], in <module>
      6 print(next(x))
      7 print(next(x))
----> 8 print(next(x))

StopIteration:
```

Ex 2

In [47]:
```
def bettermerafuctn(start,end):
    while start<end:
        yield start
        start+=1
```

In [48]:
```
for i in bettermerafuctn(3,6):
    print(i)
```

```
3
4
5
```

In [ ]:
```

```

Creating a generator fucntn to generate a no. of squares

In [36]:
```
def square(no):
    for i in range(1,no+1):
        yield i**2
```

In [38]:
```
for i in square(12):
    print(i,end=' ')
```

```
1 4 9 16 25 36 49 64 81 100 121 144
```

or

In [42]:
```
sq = square(5)

print(next(sq))
print(next(sq))

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

```
1
4

9
16
25
```

In [ ]:

Generator expression

In [62]:
```
# List comprehension
import sys

L = [i**2 for i in range(1,100)]

print(L)
print('\nsize : ',sys.getsizeof(L))
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 28
9, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 96
1, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764, 18
49, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3
025, 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]

size :  904
```

In [63]:
```
gen = (i**2 for i in range(1,100))
for i in gen:
    print(i, end=' ')

print('\nsize : ',sys.getsizeof(gen))
```

```
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 1
444 1521 1600 1681 1764 1849 1936 2025 2116 2209 2304 2401 2500 2601 27
04 2809 2916 3025 3136 3249 3364 3481 3600 3721 3844 3969 4096 4225 435
6 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 9
025 9216 9409 9604 9801
size :  112
```

In [ ]:

Benefits of generator:

```
- ease if implementation
- memory efficient
- Representing infinite streams
- chaining generators
```

In [ ]:

Chaining generators

In [85]:
```
def fibogen(no):
    f = 0
    s = 1
    for i in range(no):
        yield f
        tmp = f
        f = s
        s = s+tmp
```

In [89]:
```
for i in fibogen(5):
    print(i)
```

```
0
1
1
2
3
```

In [ ]:

In [90]:
```
def square(fiboseq):
    for no in fiboseq:
        yield no**2
```

In [94]:
```
for i in square(fibogen(5)):
    print(i)
```

```
0
1
1
4
9
```

In [ ]:

In [97]:
```
# Using chainging generators here !!

print(sum(square(fibogen(5))))
```

```
15
```

In [ ]: