In [1]:

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

In [4]:

```
#Normal function
def square_numbers(nums):
    result=[]
    for i in nums:
        result.append(i*i)
    return result
my_nums=square_numbers([1,3,4,5,6,7])
print(my_nums)
```

```
[1, 9, 16, 25, 36, 49]
```

In [9]:

```
#using generators
def square(nums):
    for i in nums:
        yield(i*i) #yield keyword
my_nums=square([1,3,4,5,6,7])
#print(next(my_nums))
#print(next(my_nums))
#print(next(my_nums))
#print(next(my_nums))
#print(next(my_nums))
for num in my_nums:
    print(num)

#this is more readable than before written program using result set
```

```
1
9
16
25
36
49
```

In [12]:

```
#By using list comprehension
nums=[x*x for x in [1,2,3,4,5]]
print(nums)

# We can also create generator like this by removing the square bracket
nums=(x*x for x in[1,2,3,4,5])
print(nums)
for num in nums:
    print(num)
```

```
[1, 4, 9, 16, 25]
<generator object <genexpr> at 0x0000018EB15C85E8>
1
4
9
16
25
```

In [13]:

```
# if we want to print out all of the values in generator like said all the values not present in the memory but w
e can
# convert to a list.
nums=(x*x for x in[1,2,3,4,5])
print(list(nums))

#Note: If you convert a generator into list, that you lose the advantages you gain interms of performance.
# Generators are very good with performance. It will not holding all values in memory.
```

```
[1, 4, 9, 16, 25]
```

In [14]:

```
#advantages of generator functions
# 1.when compared with class level iteration, 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 webscraping and crawling.
```

In [23]:

```
import random
import time
names=['aravind','rohith','sunny','bunny']
subjects=['python','java','c','c++']
def people_list(num_people):
    results=[]
    for i in range(num_people):
        person={
            'id':i,
            'name':random.choice(names),
            'subjects':random.choice(subjects)
        }
        results.append(person)
    return results
def people_generator(num_people):
    for i in range(num_people):
        person={
            'id':i,
            "names":random.choice(names),
            'subjects':random.choice(subjects)
        }
        yield(person)
'''t1=time.clock()
people=people_list(1000000)
t2=time.clock()'''

'''t1=time.clock()
people=people_generator(100000)
t2=time.clock()'''
print('took{}'.format(t2-t1))


# The execution time of generators is fast w.r.t to normal collections
```

```
took0.1616527999999562
```

In [25]:

```
# generators vs normal collections wrt memory utilization.

#Normal collection:
#l=[x*x for x in range(1000000000000)]
#print(l[0])
# We will get memory error in this case because all the values are required to store in memory.

#Generators:
g=(x*x for x in range(100000000000))
print(next(g))
print(next(g))
# We won't get any memory error because the values won't be stored at beginning.
```

```
0
1
```