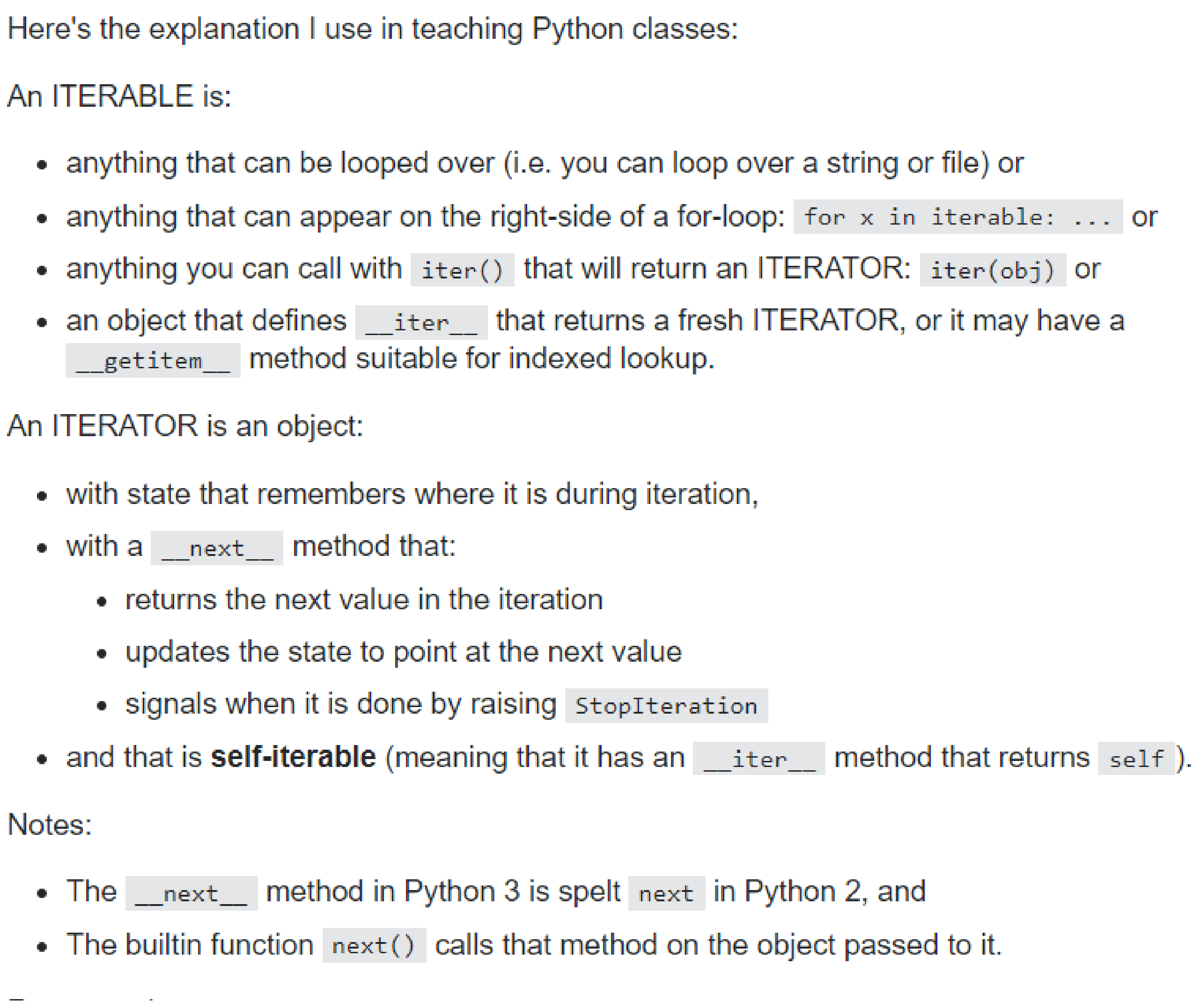
PYTHON NOTES-5

GENERATORS

CONCEPT OF ITERBALES AND ITERATORS



Here's the explanation I use in teaching Python classes:

An ITERABLE is:

- anything that can be looped over (i.e. you can loop over a string or file) or
- anything that can appear on the right-side of a for-loop: `for x in iterable: ...` or
- anything you can call with `iter()` that will return an ITERATOR: `iter(obj)` or
- an object that defines `__iter__` that returns a fresh ITERATOR, or it may have a `__getitem__` method suitable for indexed lookup.

An ITERATOR is an object:

- with state that remembers where it is during iteration,
- with a `__next__` method that:
  - returns the next value in the iteration
  - updates the state to point at the next value
  - signals when it is done by raising `StopIteration`
- and that is **self-iterable** (meaning that it has an `__iter__` method that returns `self`).

Notes:

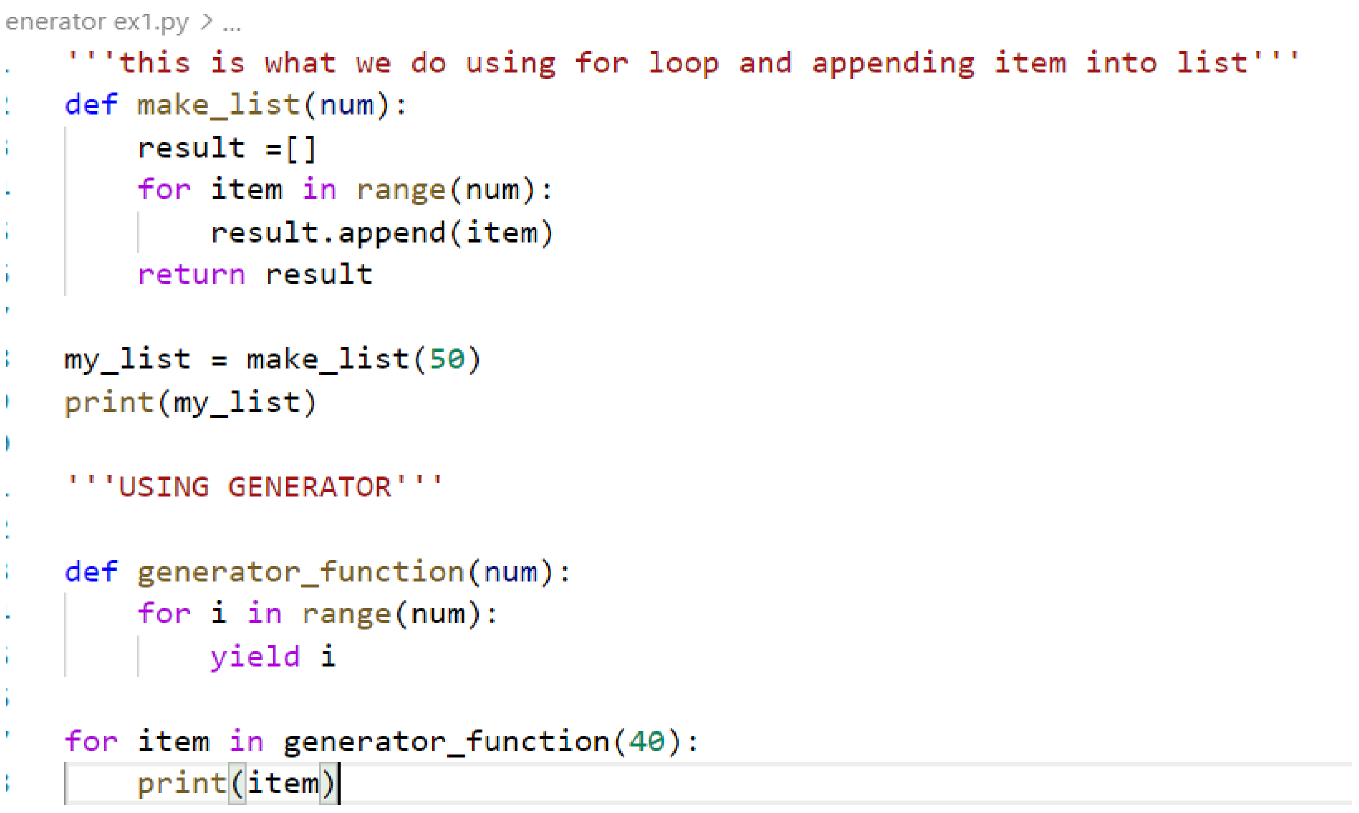
- The `__next__` method in Python 3 is spelt `next` in Python 2, and
- The builtin function `next()` calls that method on the object passed to it.

Generator-Function

- A generator-function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

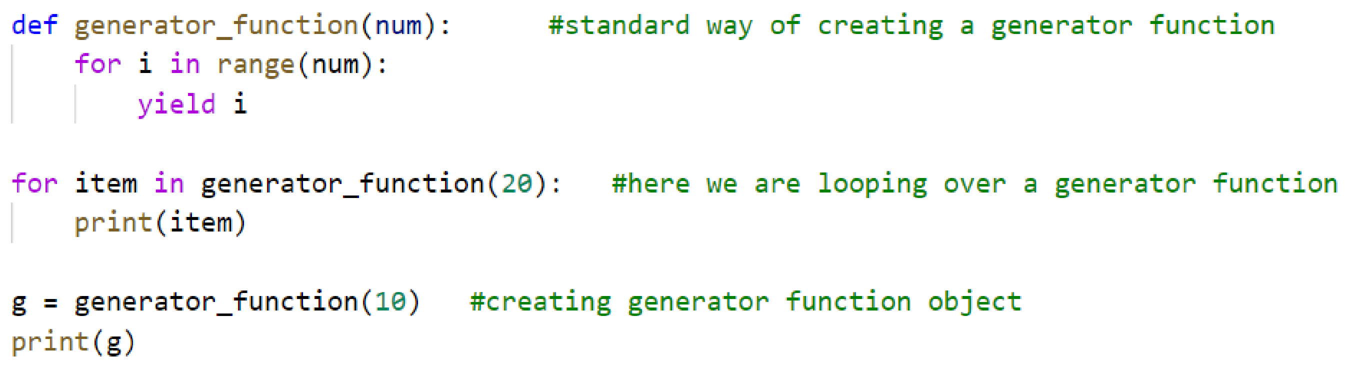
Yield Statement

- ‘yield’ pauses the function and come back to function when we do ‘next’.

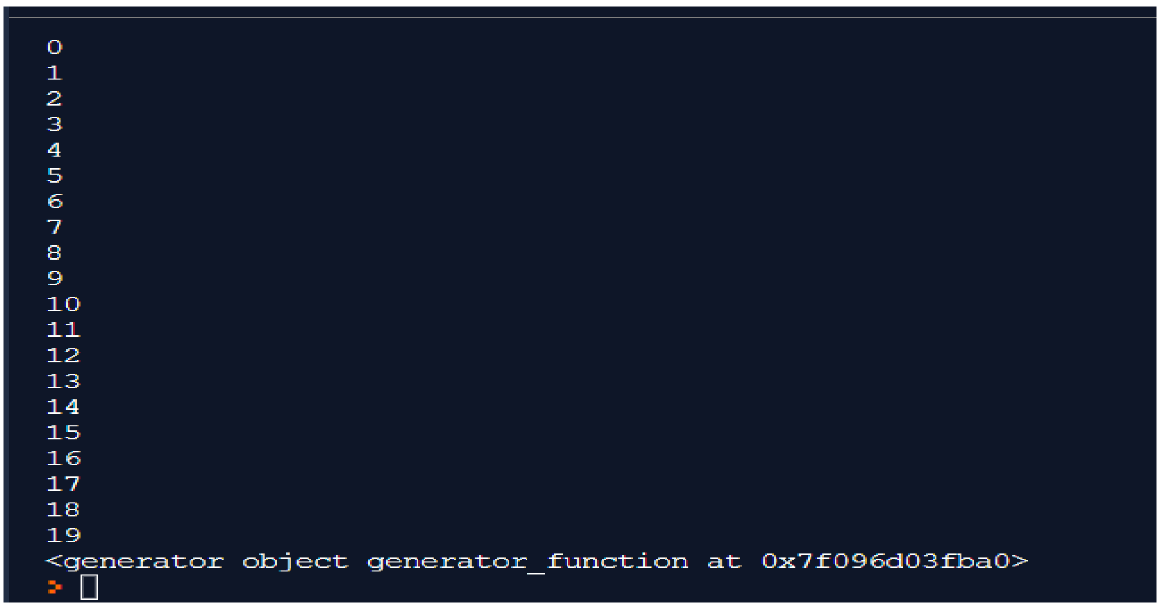


```
enerator ex1.py > ...
'''this is what we do using for loop and appending item into list'''
def make_list(num):
    result =[]
    for item in range(num):
        result.append(item)
    return result

my_list = make_list(50)
print(my_list)

'''USING GENERATOR'''

def generator_function(num):
    for i in range(num):
        yield i

for item in generator_function(40):
    print(item)
```

Creating a Generator Function and Generator Object

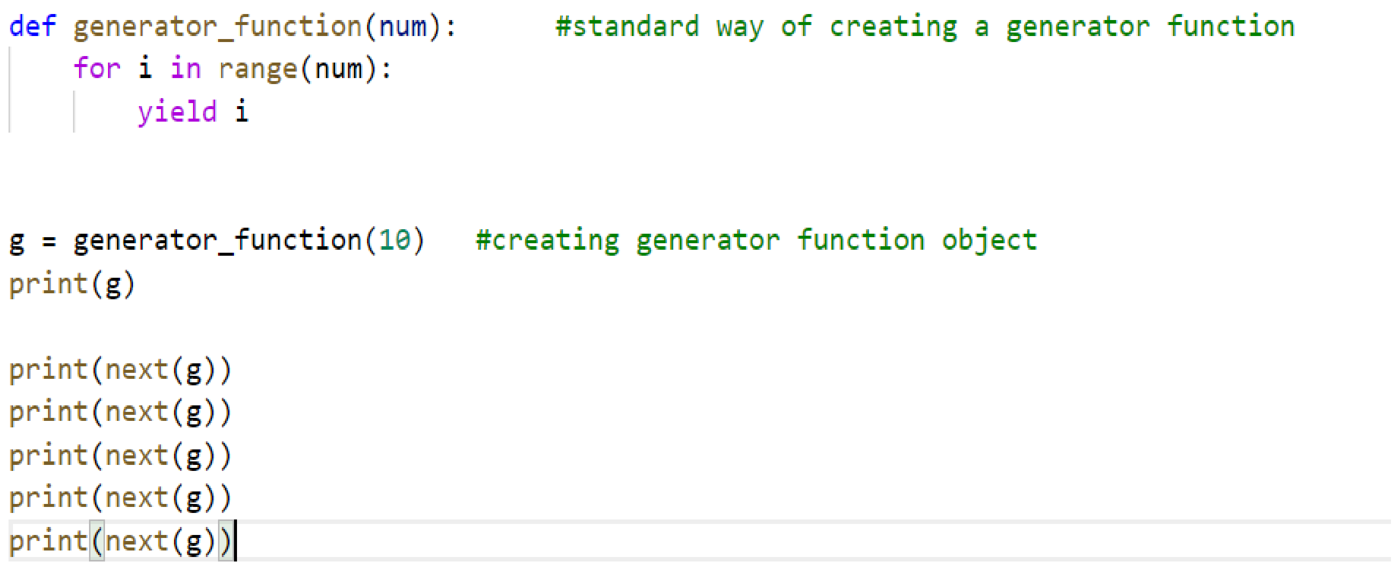


```
def generator_function(num):       #standard way of creating a generator function
    for i in range(num):
        yield i

for item in generator_function(20):    #here we are looping over a generator function
    print(item)

g = generator_function(10)    #creating generator function object
print(g)
```

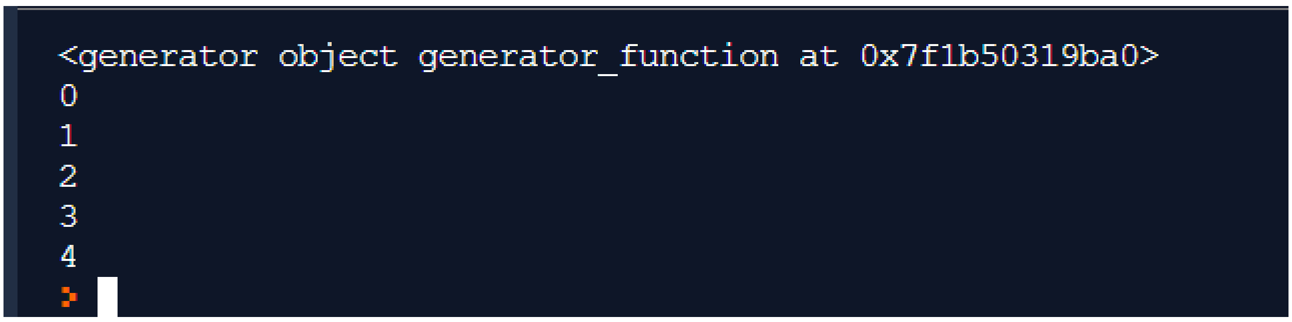


```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
<generator object generator_function at 0x7f096d03fba0>
> 
```

- Here we are creating a generator function and generator object and later on we tried generating the values using ‘next’

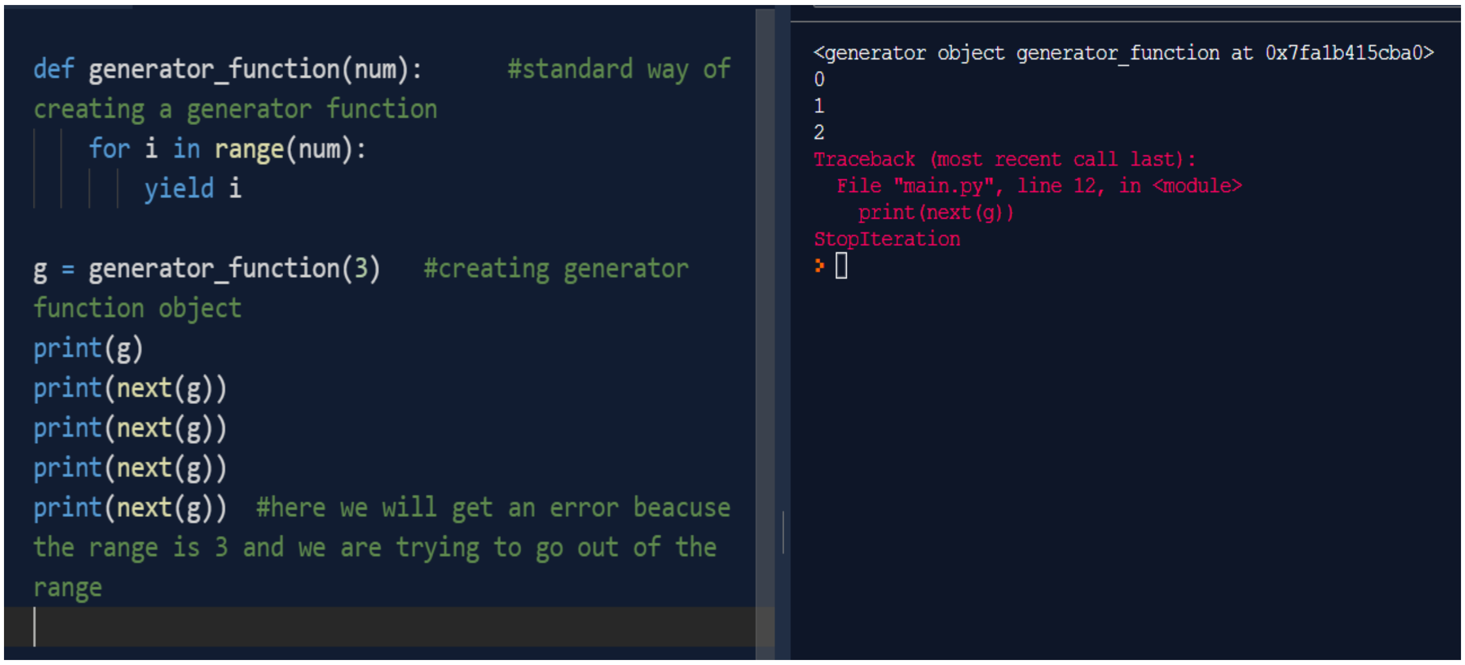


```
def generator_function(num):      #standard way of creating a generator function
    for i in range(num):
        yield i


g = generator_function(10)   #creating generator function object
print(g)

print(next(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```



```
<generator object generator_function at 0x7f1b50319ba0>
0
1
2
3
4
› 
```

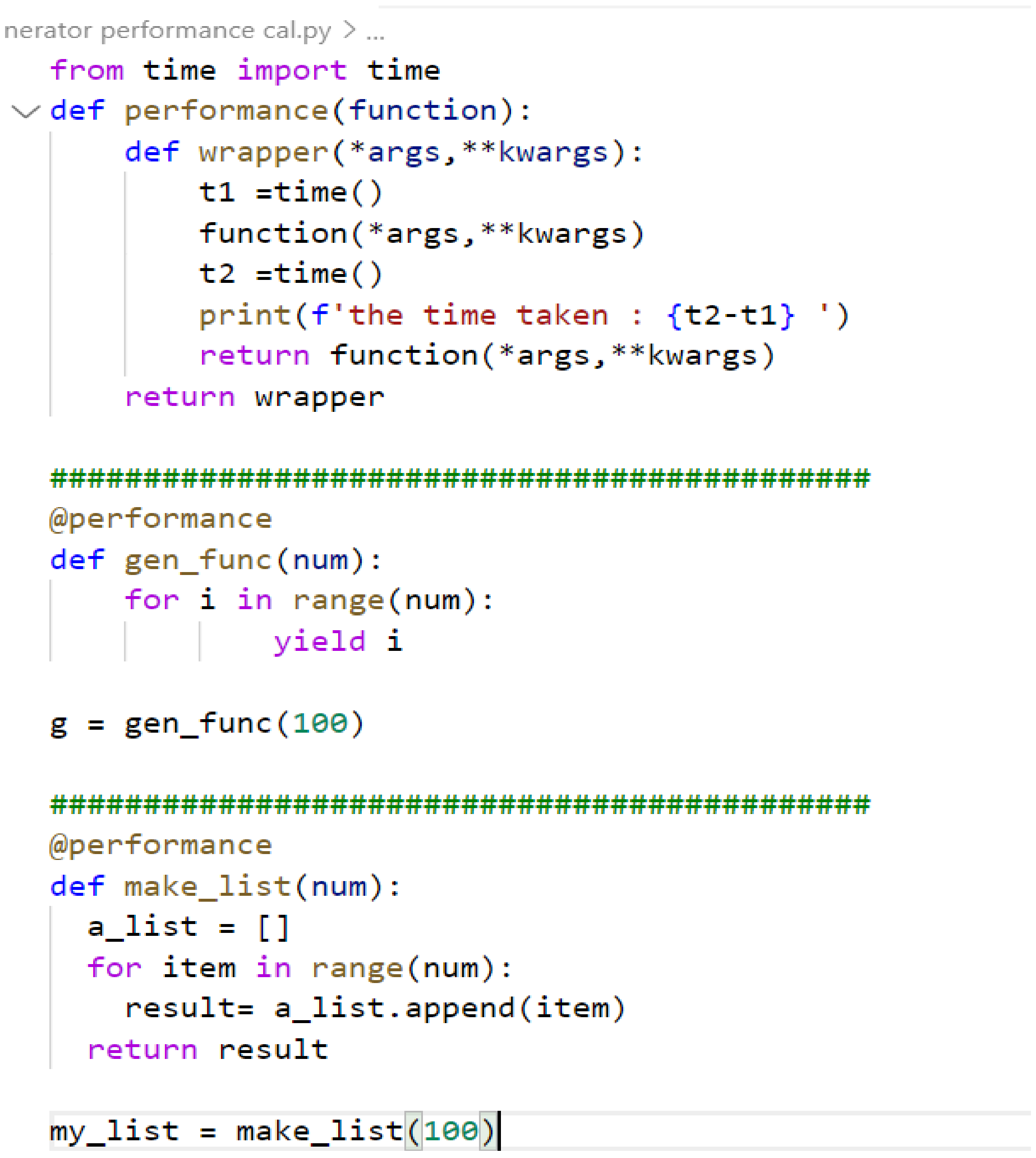
-if we try generating the values which are out of the range, the we will get *StopIteration* message. In below code, the object function we created has range 3 and we started generating values on trying printing 4th next(g), we get StopIteration message as the range was only 3 and we were going out of the range.



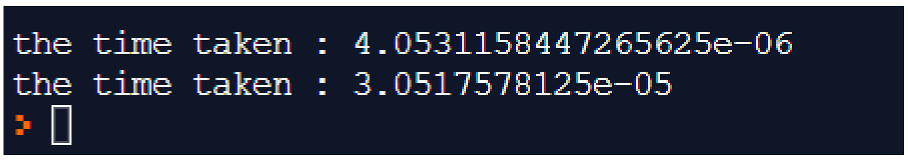
```
def generator_function(num):      #standard way of creating a generator function
    for i in range(num):
        yield i

g = generator_function(3)   #creating generator function object
print(g)
print(next(g))
print(next(g))
print(next(g))
print(next(g))  #here we will get an error beacuse the range is 3 and we are trying to go out of the range
```

```
<generator object generator_function at 0x7fa1b415cba0>
0
1
2
Traceback (most recent call last):
  File "main.py", line 12, in <module>
    print(next(g))
StopIteration
› 
```

- Generators are way faster than the list while generating the values. Here we have written the code using decorator performance for analysing the time taken by generator function and list to generate the values.



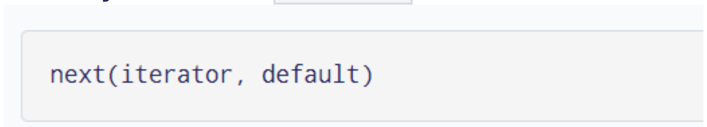
```
nerator performance cal.py > ...
from time import time
def performance(function):
    def wrapper(*args,**kwargs):
        t1 =time()
        function(*args,**kwargs)
        t2 =time()
        print(f'the time taken : {t2-t1} ')
        return function(*args,**kwargs)
    return wrapper

############################################
@performance
def gen_func(num):
    for i in range(num):
            yield i

g = gen_func(100)

############################################
@performance
def make_list(num):
  a_list = []
  for item in range(num):
    result= a_list.append(item)
  return result

my_list = make_list(100)
```



```
the time taken : 4.0531158447265625e-06
the time taken : 3.0517578125e-05
>
```

Next () function Statement

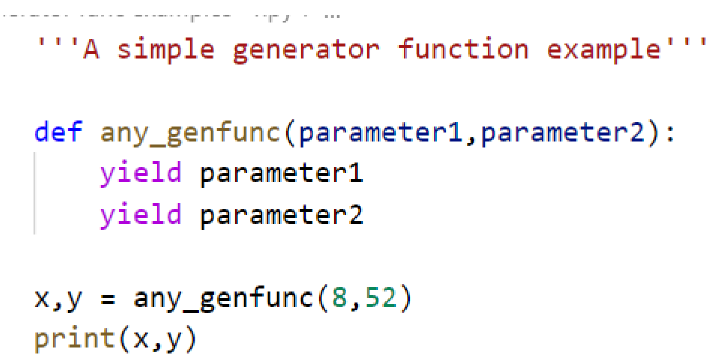
- The next () function returns the next item from the iterator.
- The syntax of `next ()` is:



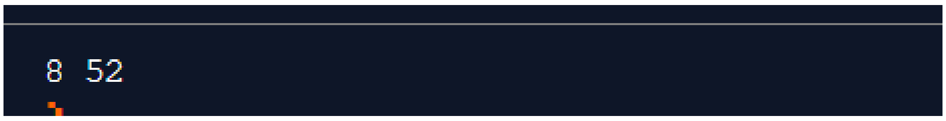
```
next(iterator, default)
```

- If the `default` parameter is omitted and the `iterator` is exhausted, it raises `StopIteration` exception.

Examples

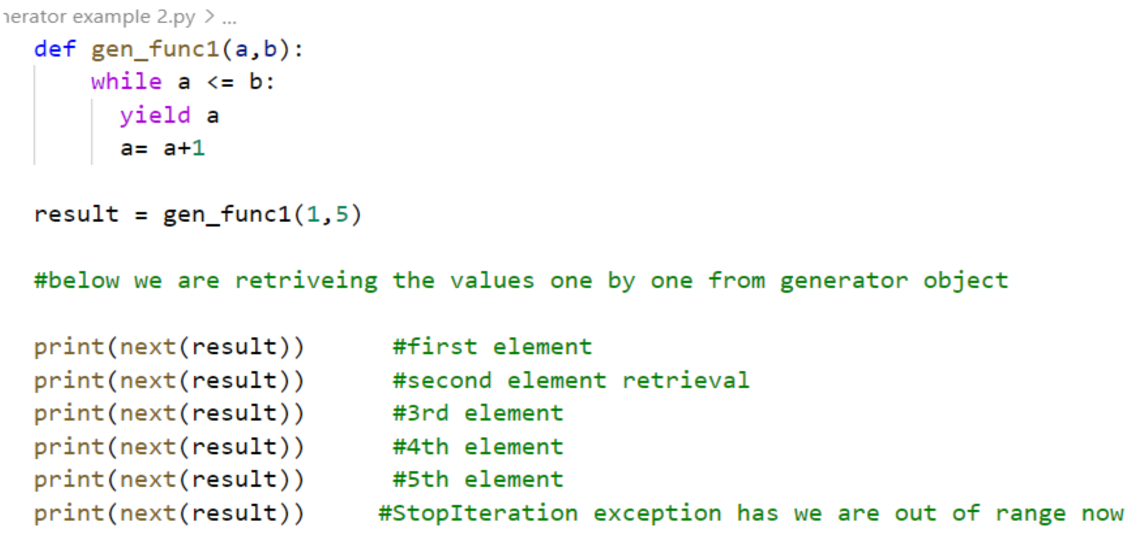


```
'''A simple generator function example'''

def any_genfunc(parameter1,parameter2):
    yield parameter1
    yield parameter2

x,y = any_genfunc(8,52)
print(x,y)
```

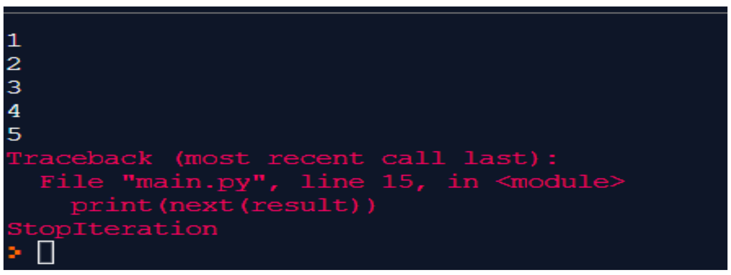


```
8 52
```

Example:

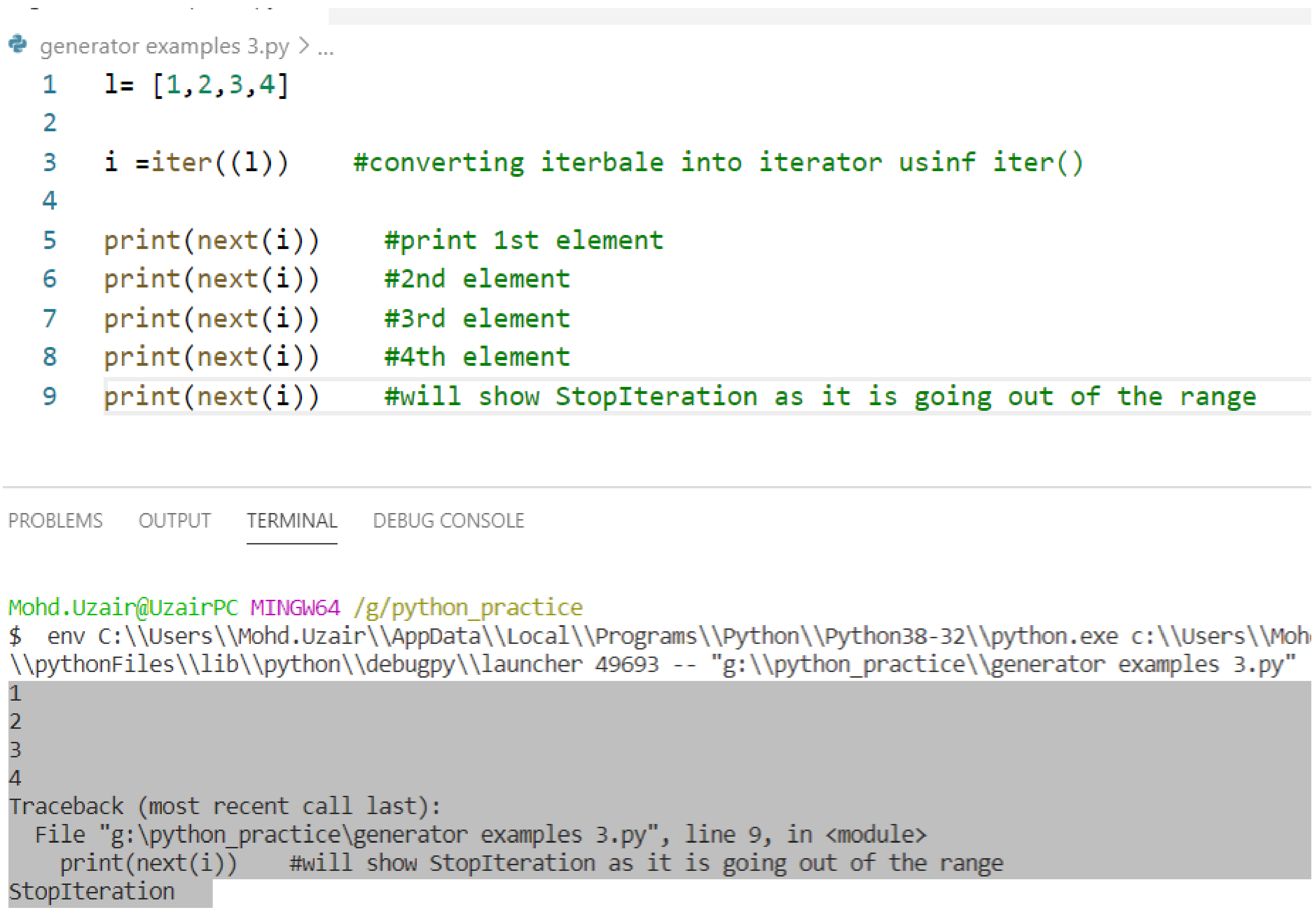


```
nerator example 2.py > ...
def gen_func1(a,b):
    while a <= b:
        yield a
        a= a+1

result = gen_func1(1,5)

#below we are retriveing the values one by one from generator object

print(next(result))      #first element
print(next(result))      #second element retrieval
print(next(result))      #3rd element
print(next(result))      #4th element
print(next(result))      #5th element
print(next(result))     #StopIteration exception has we are out of range now
```



```
1
2
3
4
5
Traceback (most recent call last):
  File "main.py", line 15, in <module>
    print(next(result))
StopIteration
> 
```

- Both generator and list are sequence, the difference is list is iterable and generator is iterator.



```python
l= [1,2,3,4]

i =iter((l))    #converting iterbale into iterator usinf iter()

print(next(i))    #print 1st element
print(next(i))    #2nd element
print(next(i))    #3rd element
print(next(i))    #4th element
print(next(i))    #will show StopIteration as it is going out of the range
```

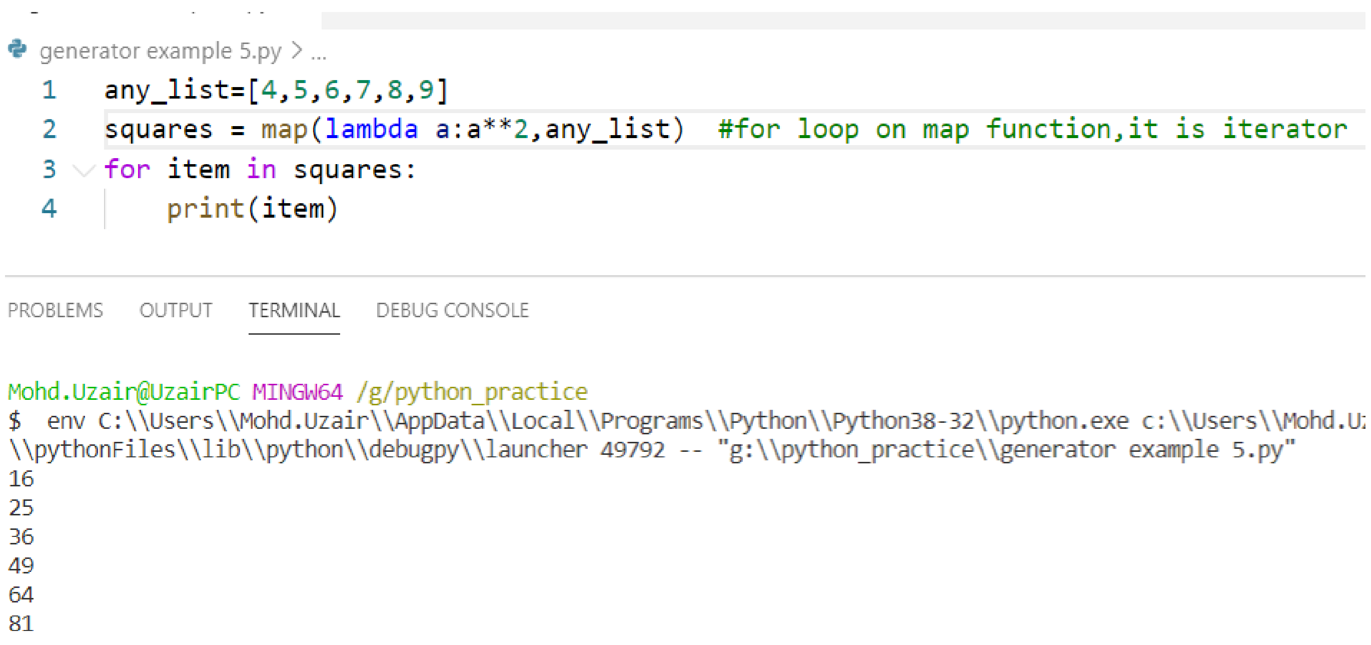
```
Mohd.Uzair@UzairPC MINGW64 /g/python_practice
$  env C:\\Users\\Mohd.Uzair\\AppData\\Local\\Programs\\Python\\Python38-32\\python.exe c:\\Users\\Moh
\\pythonFiles\\lib\\python\\debugpy\\launcher 49693 -- "g:\\python_practice\\generator examples 3.py"
1
2
3
4
Traceback (most recent call last):
  File "g:\python_practice\generator examples 3.py", line 9, in <module>
    print(next(i))    #will show StopIteration as it is going out of the range
StopIteration
```

- We cannot call next () function directly on iterable. We need to convert the iterable to iterator then we can do so. List is iterable so we first converted it into iterator using iter () function then call the next () function on it.

-

```python
any_list = [1,2,3,4,5]
next(any_list)

#since we tried calling next() function on list directly, it is giving error
'list object is not an iterator'
```

- Map function is an iterator. Below we are applying for loop on iterator function.

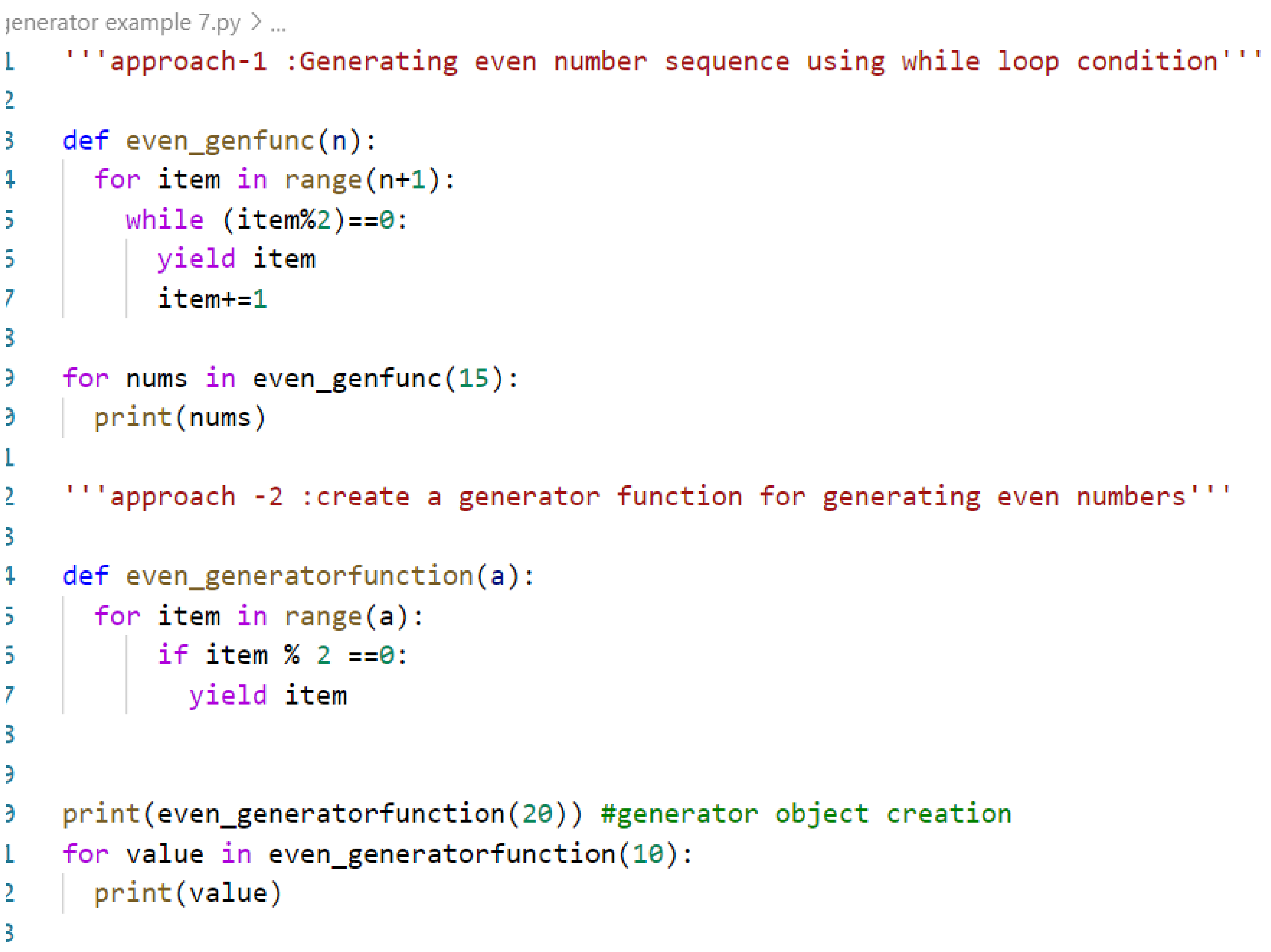

generator example 5.py > ...

```python
any_list=[4,5,6,7,8,9]
squares = map(lambda a:a**2,any_list)  #for loop on map function,it is iterator
for item in squares:
    print(item)
```

```
Mohd.Uzair@UzairPC MINGW64 /g/python_practice
$  env C:\\Users\\Mohd.Uzair\\AppData\\Local\\Programs\\Python\\Python38-32\\python.exe c:\\Users\\Mohd.U:
\\pythonFiles\\lib\\python\\debugpy\\launcher 49792 -- "g:\\python_practice\\generator example 5.py"
16
25
36
49
64
81
```

Code Exercise:

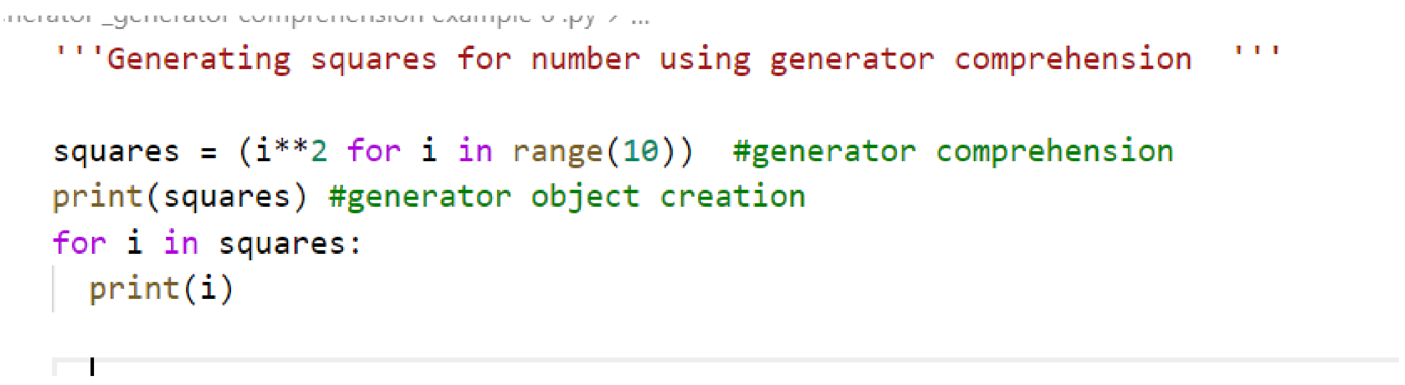
Generating even number sequence using generator function-


generator example 7.py > ...

```python
'''approach-1 :Generating even number sequence using while loop condition'''

def even_genfunc(n):
  for item in range(n+1):
    while (item%2)==0:
      yield item
      item+=1

for nums in even_genfunc(15):
  print(nums)

'''approach -2 :create a generator function for generating even numbers'''

def even_generatorfunction(a):
  for item in range(a):
      if item % 2 ==0:
        yield item


print(even_generatorfunction(20)) #generator object creation
for value in even_generatorfunction(10):
  print(value)
```

GENERATOR COMPREHENSION

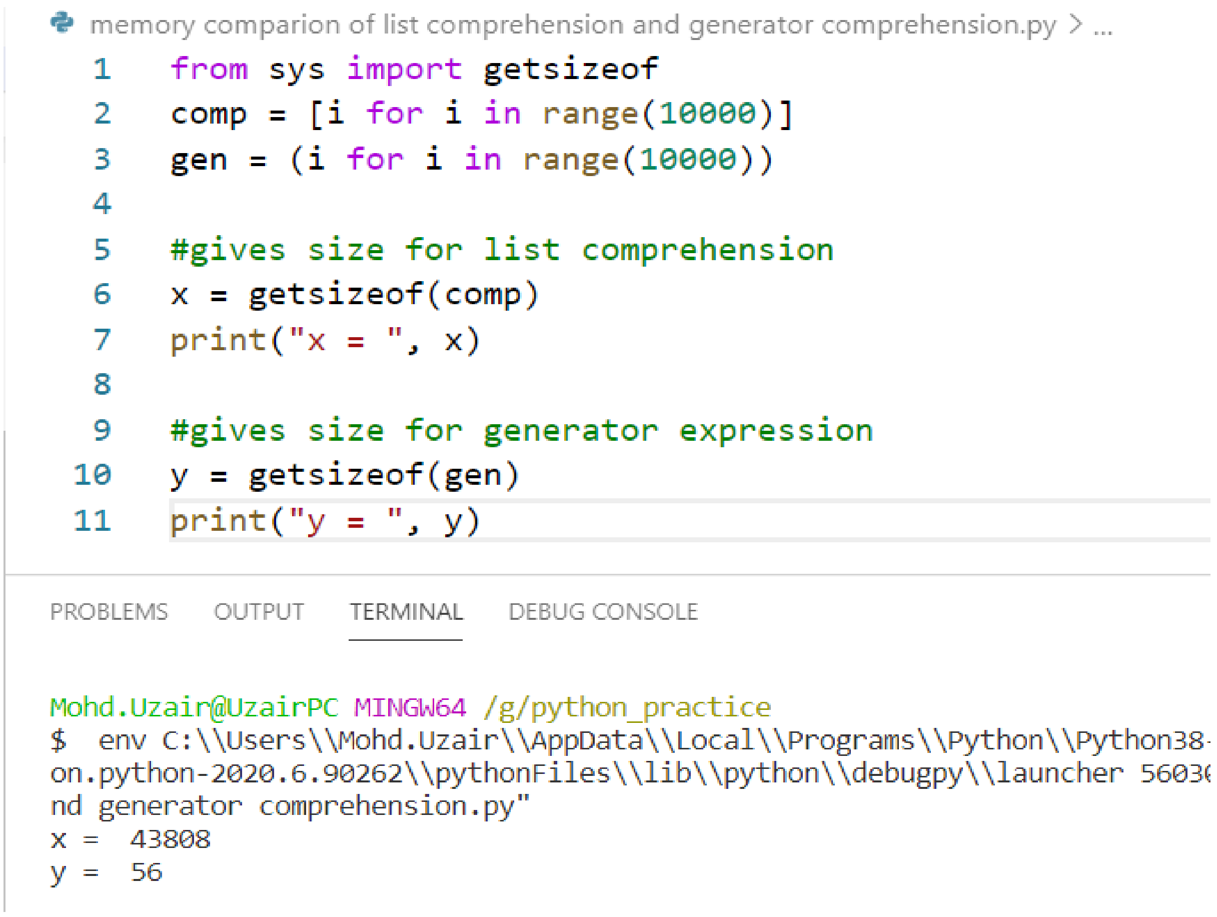
- We can generate a sequence using generator comprehension.



```
nerator _generator comprehension example 6 .py > ...
'''Generating squares for number using generator comprehension  '''

squares = (i**2 for i in range(10))  #generator comprehension
print(squares) #generator object creation
for i in squares:
    print(i)
```

- The generator yields one item at a time and generates item only when in demand. Whereas, in a list comprehension, Python reserves memory for the whole list. Thus, we can say that the generator expressions are memory efficient than the lists.
- Also, there is a remarkable difference in the execution time. Thus, generator expressions are faster than list comprehension and hence time efficient.

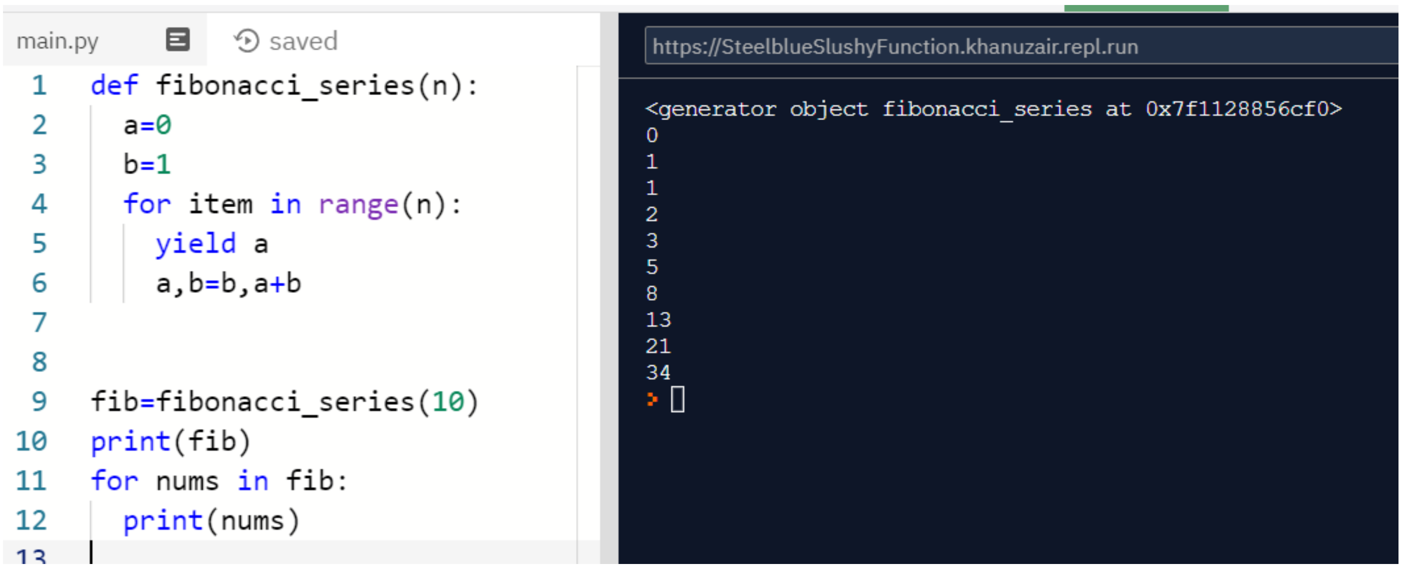
List comprehension & Generator Comprehension Memory Comparison



```
memory comparion of list comprehension and generator comprehension.py > ...
from sys import getsizeof
comp = [i for i in range(10000)]
gen = (i for i in range(10000))

#gives size for list comprehension
x = getsizeof(comp)
print("x = ", x)

#gives size for generator expression
y = getsizeof(gen)
print("y = ", y)
```

```
Mohd.Uzair@UzairPC MINGW64 /g/python_practice
$  env C:\\Users\\Mohd.Uzair\\AppData\\Local\\Programs\\Python\\Python38-
on.python-2020.6.90262\\pythonFiles\\lib\\python\\debugpy\\launcher 5603
nd generator comprehension.py"
x =  43808
y =  56
```

FIBONACCI SERIES GENERATION USING GENERATOR FUNCTION



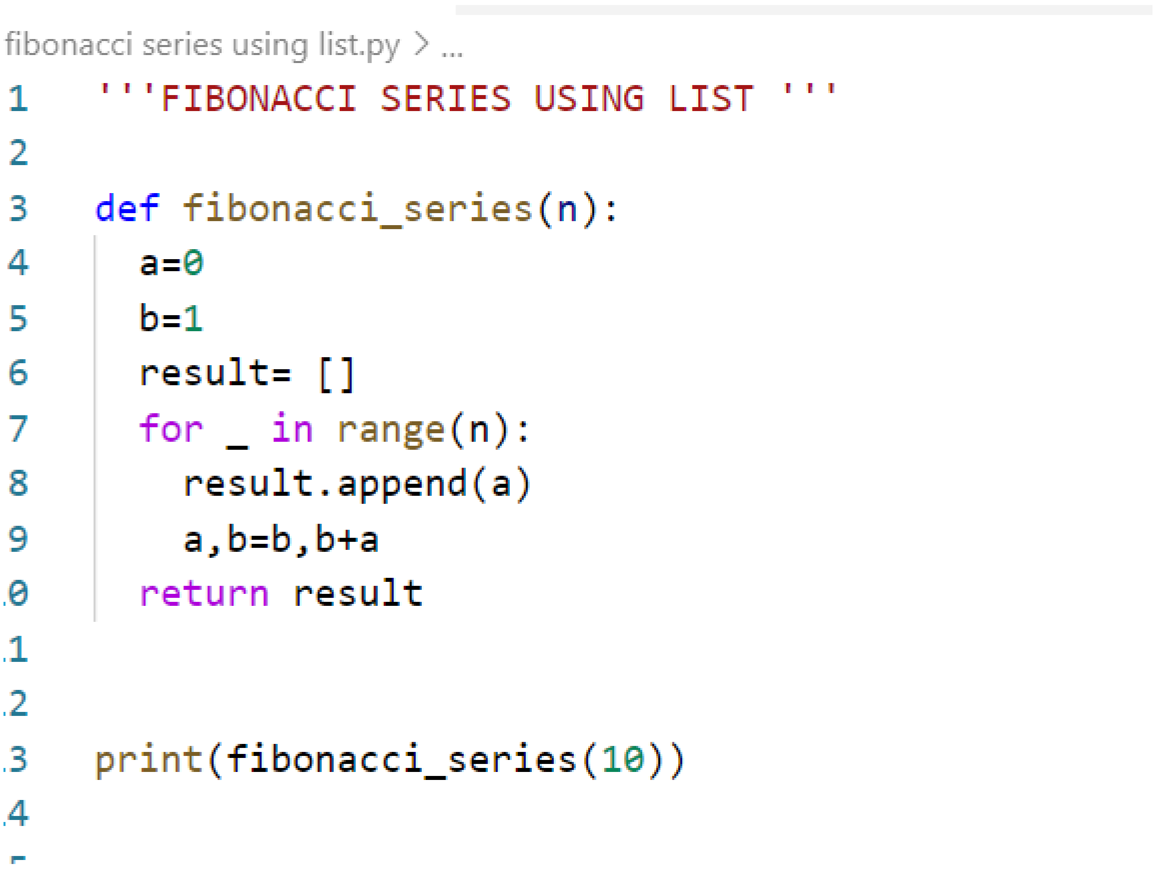
main.py saved

```python
def fibonacci_series(n):
  a=0
  b=1
  for item in range(n):
    yield a
    a,b=b,a+b


fib=fibonacci_series(10)
print(fib)
for nums in fib:
  print(nums)

```

https://SteelblueSlushyFunction.khanuzair.repl.run

```
<generator object fibonacci_series at 0x7f1128856cf0>
0
1
1
2
3
5
8
13
21
34
›
```

Other way: Fibonacci Series Using List



fibonacci series using list.py > ...

```python
'''FIBONACCI SERIES USING LIST '''

def fibonacci_series(n):
  a=0
  b=1
  result= []
  for _ in range(n):
    result.append(a)
    a,b=b,b+a
  return result


print(fibonacci_series(10))
```