Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #67
Generator

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU
&
Prof. Apoorva MS
Assistant Professor
Dept. of CSE, PESU

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M (Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Introduction

Generator is an **iterable object that has the capability to produce desired sequence of values** when iterated over it. Very helpful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once. **Generators are a great way to save memory when working with large datasets**.

There are different ways to create generator objects.

- Generator functions
- Generator expressions

Generator function:

A function which **returns an Iterator object that produces a sequence of values when iterated over it.** Iterator is discussed in detail in the next section.

Generator function **does not return a single value**. But instead **returns an Iterator object with a sequence of values.**

```
def generator_function_name(arg):
        ……………………….
          ……………………….
           ……………………….
           yield statement
```

__**Implementation:**__ It is a function **which in turn calls yield one or more times** and returns an object called the generator**. No statement in the function is executed** when called **if the function contains yield** statement within it. **Example_code_4 demonstrates this.**

Key points about Generators:

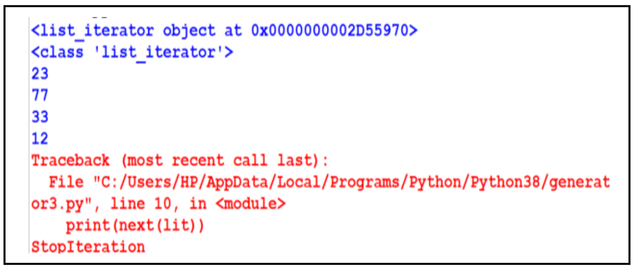
- The generator function has one or more yields.
- The generator function as well as the code calling the next(), stays in some state of execution simultaneously. This concept is called co-routine.
- The generator function does not execute first time it is called – instead returns a generator object.
- The generator function resumes from where it had left of in the earlier execution of call on the generator object.
- The generators are lazy . They do not produce all the results at a time.

Let us understand the **concept of Iterator** before digging into Generator function. **Iterators are objects that allow you to traverse through all the elements of a collection** and return one element at a time. Some examples of iterables are Lists, Tuples, Strings, Dictionaries, Sets for which iterators can be created.

Consider,

Example_code_1:

```
li = [23,77,33,12]   # li is iterable
#lit = li.__iter__()    #creation of iterator object using __iter__. This is a specific function in list
lit = iter(li)             #creation of iterator object using iter(). Available in builtins
print(lit)
print(type(lit))   #observe this.
print(next(lit))
print(next(lit))
print(next(lit))
print(next(lit))
print(next(lit))
```

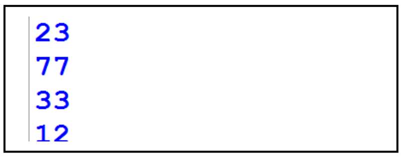


```
<list_iterator object at 0x0000000002D55970>
<class 'list_iterator'>
23
77
33
12
Traceback (most recent call last):
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/generat
or3.py", line 10, in <module>
    print(next(lit))
StopIteration
```

When next() is called on the iterator object, it returns one element from the iterator object and removes that element from the iterator object. When there are no more elements in the Iterator object, it returns an exception – stopIteration

Example_code_2: Responsibilities of for loop is demonstrated in this example

```
li = [23,77,33,12]   # li is iterable
for item in li:
        print(item)
```



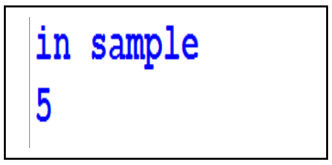
```
23
77
33
12
```

The **for loop** will automatically call the **iter() method** on the iterable object, which will **return an iterator object**. The for loop will then iterate over the iterator object, calling **the next()** method on it **to get the next element**. The next() method will **raise a StopIteration exception** when there are no more elements and this exception is successfully handled by the for loop automatically.

Now consider the below simple code.

Example_code_3:

```
def sample():
    print("in sample")
    return 5
    print("in sample1")
    return 6
    print("in sample2")
    return10
g1 = sample()
print(g1)
g2 = sample()
print(g2)
```

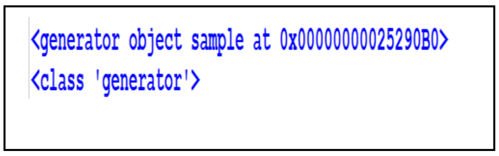


```
in sample
5
```

Observe that the function sample is a normal function. It doesn't pause the execution of the function. Each time the function is run, value 5 is returned. Even if the function contains many statements after the first return statement, they are unreachable. How do we pause and resume from within the function? Use generator function.

Example_code_4: First example of generator function

```
def sample():
    print("in gen1")
    yield 5
    print("in gen1 after first yield")
    yield 6
    print("in gen1 after second yield")
    yield 10
g = sample()
print(g)
print(type(g))
```



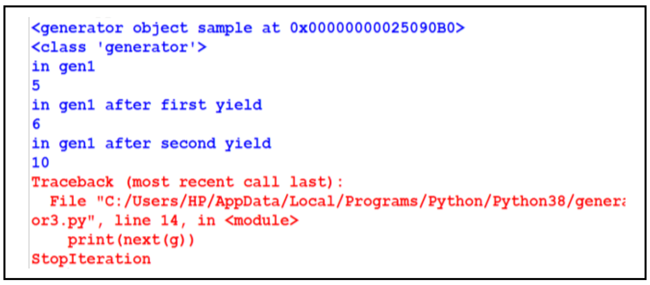
```
<generator object sample at 0x00000000025290B0>
<class 'generator'>
```

When the statements in the definition of sample() will be executed then? Statements within the generator function will be executed when you call next() function on the generator object. Refer to Example_code_5.

Can you use for loop to iterate the elements of the generator object? Yes, it is iterable. Refer to Example_code_6.

Example_code_5:

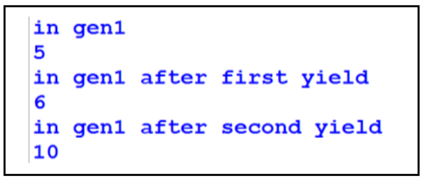
```
def sample():
    print("in gen1")
    yield 5
    print("in gen1 after first yield")
    yield 6
    print("in gen1 after second yield")
    yield 10
g = sample()
print(g)
print(type(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))   # Think about what this prints?
```



```
<generator object sample at 0x00000000025090B0>
<class 'generator'>
in gen1
5
in gen1 after first yield
6
in gen1 after second yield
10
Traceback (most recent call last):
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/genera
or3.py", line 14, in <module>
    print(next(g))
StopIteration
```

Example_code_6:

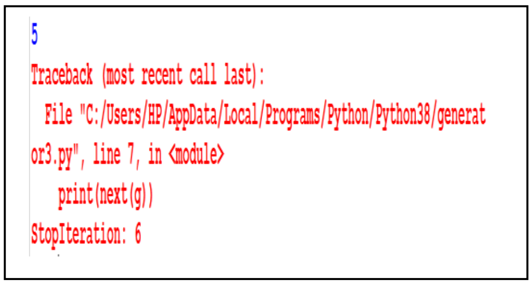
```
g = sample()
for item in g:          # prints all items from g.
        print(item) #  running one more for loop
                #again after this loop .
```



```
in gen1
5
in gen1 after first yield
6
in gen1 after second yield
10
```

Example_code_7:

```
def gen1():
    yield 5
    return 6
    return 10
g = gen1()
print(next(g))
print(next(g))
print(next(g))
```



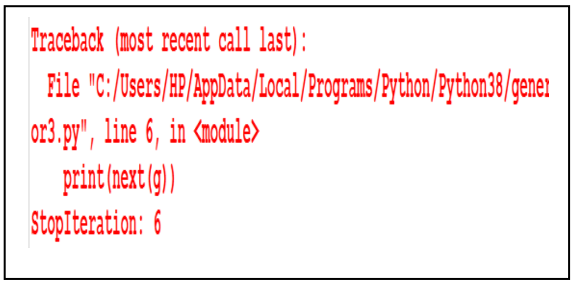
```
5
Traceback (most recent call last):
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/generat
or3.py", line 7, in <module>
    print(next(g))
StopIteration: 6
```

Few points to think about Example_code_7:

- Is it generator function?
- If yes, how many items are there in the iterator object returned by the generator function?
- If no, why?
- How many times the above code prints 6 and 10?
- If yes, How many times Exception is thrown in the output?

Example_code_8: Is this generator? Think about the answers for above mentioned points for this example.

```
def gen1():
    return 6
    yield 5
    return 10
g = gen1()
print(next(g))
print(next(g))
print(next(g))
```



```
Traceback (most recent call last):
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/gener
or3.py", line 6, in <module>
    print(next(g))
StopIteration: 6
```

Generator Expression:

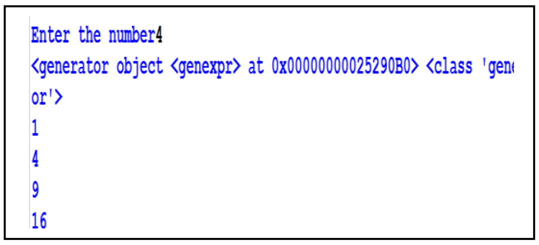
This is an another way of writing the generator function **similar to List comprehension[details are covered in Unit-4].** But **instead of storing the elements in a list in memory, it creates generator objects.**

Syntax: (expression for element in iterable) #(and) very important. It is not tuple here

Let us see few example codes.

Example_code_9: Print the squares of numbers from 1 to n in separate lines.

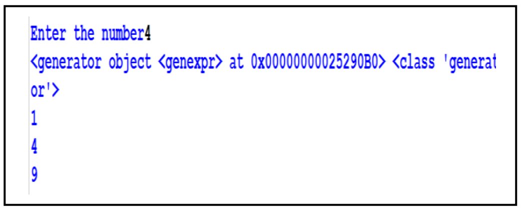
```
n = int(input("Enter the number"))
g = (i*i for i in range(1, n+1))
print(g, type(g))
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```



```
Enter the number4
<generator object <genexpr> at 0x00000000025290B0> <class 'gene
or'>
1
4
9
16
```

Example_code_10: Possible to use for loop? Yes. As discussed above.

```
n = int(input("Enter the number"))
g = (i*i for i in range(1, n+1))
print(g, type(g))
for i in g:
    print(i)
```



```
Enter the number4
<generator object <genexpr> at 0x00000000025290B0> <class 'generat
or'>
1
4
9
```

Example_code_11: Square of a number between 0 and 5 if the number is even

```
generator_exp = (i**2 for i in range(5) if i%2==0)
for i in generator_exp:
        print(i)
```

Output:

```
0
4
16
```

Pipelining Generators: Multiple generators can be used to pipeline a series of operations

Let us consider an example to compute **the sum of squares of numbers in the Fibonacci series.**

Example_code_12: We will code this step by step.

- The first step is to write a generator function which generates the Fibonacci series.
- Next step is to write a generator function which generates the squares of numbers
- Combining these two so that output of first generator object is sent as an input to second generator function to get the required result.
- Then once we get the squares, perform the sum operation on it.

Step1:

```
def generate_fib(nums):
   x,y=0,1  # First two numbers in the Fibonacci series are defined already
   for i in range(nums):
      yield x
      x,y=y,x+y
n = int(input("Enter the number of elements you want to generate in the fib series"))
gen_fib = generate_fib(n)
for item in gen_fib:
         print(item, end = " ")
```

Step 2:

```
def generate_square(nums):
   for num in range(nums):
      yield num**2
n = int(input("Enter the number of elements you want the squares -> starting from 0"))
gen_sqrs = generate_square(n)
for item in gen_sqrs:
         print(item, end = " ")
```

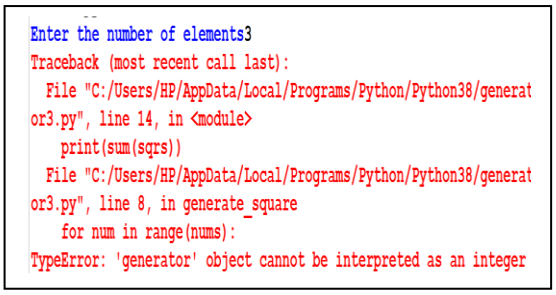
Both of the above generator functions are absolutely working fine when we separately execute these. Now we shall combine these two to fulfill our requirement. Refer to step3 and step4 codes.

Step 3 & step 4:

```
def generate_fib(nums):
   x,y=0,1
   for i in range(nums):
      yield x
      x,y=y,x+y

def generate_square(nums):
   for num in range(nums):
      yield num**2

n = int(input("Enter the number of elements"))
fibs = generate_fib(n)
sqrs = generate_square(fibs)
print(sum(sqrs))
```



```
Enter the number of elements3
Traceback (most recent call last):
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/generat
or3.py", line 14, in <module>
    print(sum(sqrs))
  File "C:/Users/HP/AppData/Local/Programs/Python/Python38/generat
or3.py", line 8, in generate_square
    for num in range(nums):
TypeError: 'generator' object cannot be interpreted as an integer
```

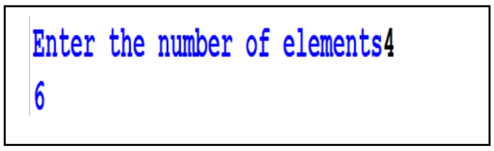
The exception says generator object cannot be interpreted as an integer. Here, fibs is a generator object which is passed to generate_square function. Fibs is copied to nums – pass by value. Inside the function, range(nums) is throwing this exception as range expects integer only. Can we just use the generator object in the loop directly?

Modified step 3 and step 4:

```
def generate_fib(nums):
   x,y=0,1
   for i in range(nums):
      yield x
      x,y=y,x+y

def generate_square(nums):
   for num in nums:    #modified only this as generator object is iterable
      yield num**2

n = int(input("Enter the number of elements"))
fibs = generate_fib(n)
sqrs = generate_square(fibs)
print(sum(sqrs))
```



```
Enter the number of elements4
6
```

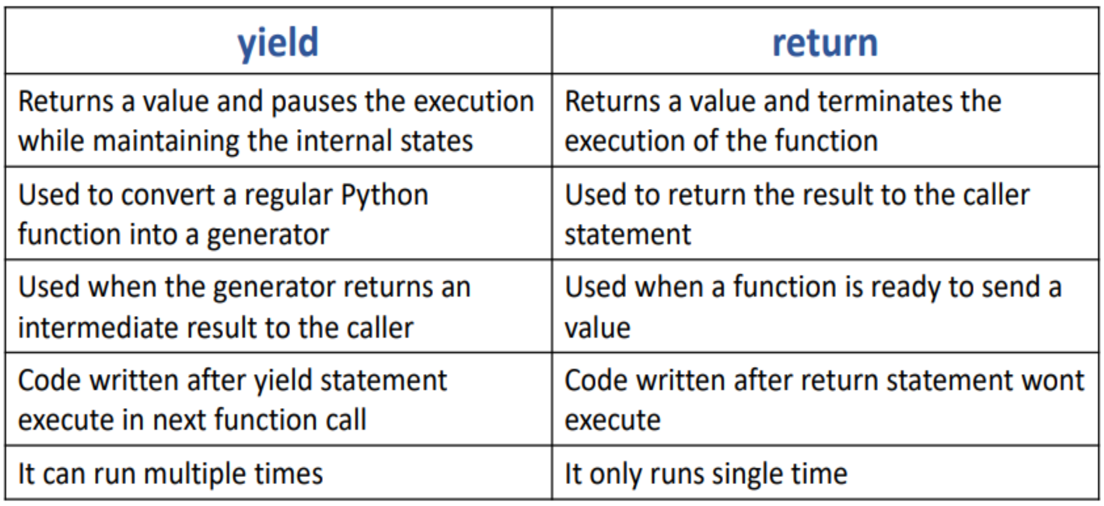
Can we pipeline these functions in one line? Yes

```
n = int(input("Enter the number of elements"))
print(sum(generate_square(generate_fib(n))))
```

Benefits of Generators:

- Supports lazy evaluation: Memory is saved as the items are produced when required.
- Generators are really nice tool to express certain ideas in a very clean and concise fashion.

Differences between yield and return:



yield	return
Returns a value and pauses the execution while maintaining the internal states	Returns a value and terminates the execution of the function
Used to convert a regular Python function into a generator	Used to return the result to the caller statement
Used when the generator returns an intermediate result to the caller	Used when a function is ready to send a value
Code written after yield statement execute in next function call	Code written after return statement wont execute
It can run multiple times	It only runs single time

-END–