WHAT ARE GENERATORS?

* Generators:
  + Are functions that return traversable objects.
  + Produce items one at a time and only when required.
  + Are run along with ‘for’ loops.

ADVANTAGES OF USING GENERATORS:

1. Easy to implement (implement\_iter\_().\_next\_() automatically
2. Better memory management and utilization
3. Can be produce infinite items
4. Can also pe used to pipeline a number of operations.

NORMAL FUNCTIONS VS GENERATORS:

|  |  |
| --- | --- |
| Generator Functions | Normal Functions |
| Make use of ‘yield’ keyword | Make use of ‘return’ keyword |
| Run when ‘next()’ method is called | Run when name of the method is called |
| Produce items one at a time and only when required | Produce all the items at once |

WRITING GENERATORS IN PYTHON:

* Generators created using ‘def’ keyword.
* Make use of yield keyword instead of return.
* Syntax:
  + def func(a):

yield a

a = [1,2,3]

b = func(a)

next(b)

* Example:
  + def new(dict):

for x,y in dict.items():

yield x,y

a = {1:"HI", 2: "wELCOME"}

b = new(a)

print(b)

next(b)

* + Output: <generator object new at 0x0000028DD80CD048>

(1, 'HI')

* + next(b)
  + Output: (2, 'wELCOME')
  + next(b)
  + Output: StopIteration:
* Example:
  + def myfunc(i):

while i<=3:

yield i

i+=1

j = myfunc(2)

next(j)

Output: 2

next(j)

Output: 3

next(j)

Output: StopIteration

* Example:
  + def ex():

n = 3

yield n

n = n\*n

yield n

y = ex()

next(y)

* + Output: 3
  + next(y)
  + Output: 9
  + next(y):
  + Output: StopIteration

GENERATORS WITH LOOPS:

* To execute generator function at once, you can use ‘for loop’. This loop iterates over all the objects and after all implementations, it executes StopIteration.
* Syntax:
  + def z():

n = 1

yield n

n = n + 3

yield n

for x in z():

print(x)

* Output: 1

4

* Example:

def ex():

n = 3

yield n

n = n\*n

yield n

y = ex()

for x in y:

print(x)

* Output: 3

9

Generator Expressions:

* Resemble list comprehensions and like lambda functions. Generator expressions create anonymous generator functions.
* a = range(6)

print(“Generator Expression”, end= ‘:n’)

c = (x+2 for x in a)

print (c)

print(min( c ))

* Example:
  + f = range(6)

print("List Comp", end=":")

q = [x+2 for x in f]

print(q)

print("Gen Comp", end=":")

r = (x+2 for x in f)

print(r)

Output: List Comp:[2, 3, 4, 5, 6, 7]

Gen Comp:<generator object <genexpr> at 0x0000028DD81655C8>

* Example:
  + for x in r:

print(x)

Output: 2

3

4

5

6

7

* Example:

print("gen exp", end=":")

r = (x+2 for x in f)

print(r)

print(min(r))

Output: gen exp:<generator object <genexpr> at 0x0000028DD81654C8>

2

USE CASES

* Fibonacci Series
  + A series of numbers where in each number also called as the Fibonacci Number is the sum of the two preceding numbers.
  + Example:
    - def fib():

f,s=0,1

while True:

yield f

f,s=s, f+s

for x in fib():

if x > 50:

break

print(x, end=" ")

* + - Output: 0,1,1,2,3,5,8,13,21,34,55
* Number Stream
  + Generating stream of numbers
  + Example:

a = range(100)

b = (x for x in a)

print(b)

for y in b:

print(y)

Output: 0 to 99 numbers

* Example:
  + a = range(2,100,2)

b = (x for x in a)

print(b)

for y in b:

print(y)

Output: Even Numbers

* Example:
  + a = range(2,100,2)

b = (x for x in a)

print(b)

for y in b:

print(y)

* + Output: Odd Numbers
* Sinewave
  + Generating sine waves using Seaborn
  + Example:
    - import numpy as np
    - from matplotlib import pyplot as plt
    - import seaborn as sb
    - def s(flip = 2):
    - x = np.linspace(0 , 14 , 100)
    - for i in range(1,5):
    - plt.plot(x, np.sin(x + i \* .5) \* (7 - i) \* flip)
    - sb.set()
    - s()
    - plt.show()
  + Example:
    - import numpy as np
    - from matplotlib import pyplot as plt
    - import seaborn as sb
    - def s(flip = 2):
    - x = np.linspace(0 , 14 , 100)
    - for i in range(1,10):
    - yield(plt.plot(x, np.sin(x + i \* .5) \* (7 - i) \* flip))
    - sb.set()
    - s = s()
    - plt.show()
    - print(next(s))