# **Generators - CMS**

**def** square\_num(lists):  
 result = []  
 **for** num **in** lists:  
 result.append(num \*\* 2)  
 **return** result

print(square\_num([1, 2, 3, 4, 5])) *# [1, 4, 9, 16, 25]*

*\_\_\_\_\_\_\_\_\_\_\_*

*#* ***Generators*** *- generator doesn’t hold the entire result in memory, rather it yields one result at a time and waiting us to ask for the next() result.*

*# Generators are iterators as well but the \_\_iter\_\_() and \_\_next\_\_() methods are created automatically.*

**def** square\_num2(lists):  
 **for** num **in** lists:  
 **yield** num \*\* 2

res = square\_num2([6, 7, 8, 9, 10])

print(res) # <generator object square\_num at 0x0000015FD30615E8>

print(next(res)) # 36

print(next(res)) # 49

print(next(res)) # 64

print(next(res)) # 81

print(next(res)) # 100

**for** i **in** res:  
 print(i)

# 36

# 49

# 64

# 81

# 100

*\_\_\_\_\_\_\_\_\_\_\_*

# using list comprehension

square\_num3 = [num \*\* 2 for num in [11, 12, 13, 14]]

print(square\_num3) # [121, 144, 169, 196]

for num in square\_num3:

print(num)

# 121

# 144

# 169

# 196

*\_\_\_\_\_\_\_\_\_\_\_*

# using generator expression

# list compreh.. are wrapped with **`[]`**, generators are wrapped with **`()`**

square\_num3 = (num \*\* 2 for num in [11, 12, 13, 14])

print(square\_num3) # <generator object <genexpr> at 0x000001FBB36D25E8>

# in order to convert generator `square\_num3` to list, use **list()** method.

print(list(square\_num3)) # [121, 144, 169, 196]

for num in square\_num3:

print(num)

# 121

# 144

# 169

# 196

## Memory Performance of **lists vs generators**

## **Using List**

*# since* ***list holds the entire result in returned value at a time****, it takes lots of memory space in contrast to generators the returns one value at a time and take less memory space.*

import memory\_profiler as mem\_profile

import random, time

names = ['John', 'Lucy', 'Mike', 'Rozzy', 'Daniel', 'Steward', 'Stephen']

majors = ['CIA', 'DEA', 'FBI', 'Software Engineer', 'Djangonaut]

print('Ram memory before function call {}'.format(mem\_profile.memory\_usage()))

def people\_list(num\_vals):

results = []

for i in range(num\_vals):

person = {

'id': i,

'name': random.choice(names),

'majors': random.choice(majors)

}

results.append(person)

return results

t1 = time.time()

res = people\_list(1000000)

t2 = time.time()

print('Ram memory after function call {}'.format(mem\_profile.memory\_usage()))

print('Estimated time: {} '.format(t2-t1))

# Ram memory before function call [37.5546875]

# Ram memory after function call [308.6875]

# Estimated time: 6.661834478378296

## **Using Generators**

## *# An important reason why to use generators is that it takes less memory space since it stores one value at a time and waits for user to ask the next value.*

*#* ***Generators hold one value at a time and not the entire value (result)****, it takes extremely less memory space in contrast to lists.*

*# when dealing with millions of data it is prefered to use generators since it takes less memory.*

import memory\_profiler as mem\_profile

import random, time

names = ['John', 'Lucy', 'Mike', 'Rozzy', 'Daniel', 'Steward', 'Stephen']

majors = ['CIA', 'DEA', 'FBI', 'Software Engineer', 'Doctor', 'Engineer', 'UFC fighter']

print('Ram memory be4 function call {}'.format(mem\_profile.memory\_usage()))

def people\_generator(num\_vals):

for i in range(num\_vals):

person = {

'id': i,

'name': random.choice(names),

'majors': random.choice(majors)

}

yield person

t1 = time.time()

res = people\_generator(1000000)

print(next(res))

t2 = time.time()

print('Ram memory after function call {}'.format(mem\_profile.memory\_usage()))

print('Estimated time: {} '.format(t2-t1))

# Ram memory before function call [37.484375]

# {'id': 0, 'name': 'Steward', 'majors': 'FBI'}

# Ram memory after function call [37.48828125]

# Estimated time: 0.0

*\_\_\_\_\_\_\_\_\_\_\_*

*# Telusku - multiple yields*  
def top\_ten():

yield 1 # the first time you call next() the function stops from here

yield 2 # Next time it stops here

yield 3 # Next time it stops here

yield 4 # Next time it stops here

res = top\_ten()

print(res.\_\_next\_\_())

print(res.\_\_next\_\_())

print(res.\_\_next\_\_())

print(res.\_\_next\_\_())

# 1

# 2

# 3

# 4

*\_\_\_\_\_\_\_\_\_\_\_*

<https://stackoverflow.com/questions/41191412/no-module-named-mem-profile>

*\_\_\_\_\_\_\_\_\_\_\_*

### [No module named mem\_profile](https://stackoverflow.com/questions/41191412/no-module-named-mem-profile)

Do the following :

* In terminal, run $ pip install memory\_profiler
* In your script, replace import mem\_profile with import memory\_profiler as mem\_profile
* replace **all** mem\_profile.memory\_usage\_resource() with mem\_profile.memory\_usage()

# Amuls Academy

Generator function is a function which returns generator-iterator with the help of yield keyword.

generator-iterator means that generator function returns a generator object which is generator iterator or in simple words (generator object = generator-iterator)

def num\_printer(max\_val):

a, b = 0, 1

while True:

c = a + b

if c < max\_val:

print('Before \'yield\' keyword')

# the first time you call next() the function stops from here

yield c

# as you call other next() it restarts from here on

print('After \'yield\' keyword')

a = b

b = c

else:

break

gen = num\_printer(10)

print(gen) # This is generator iterator that you can iterate upon it

# <generator object fib at 0x00000216F75E25E8>

print(gen.\_\_next\_\_())

# Before 'yield' keyword

# 1

print(gen.\_\_next\_\_())

# After 'yield' keyword

# Before 'yield' keyword

# 2

# Pylenin

You can have multi yield keywords in your function

def multi\_yield():

name = 'Rozzy'

yield name

name = 'Blaze'

yield name

res = multi\_yield()

print(res.\_\_next\_\_()) # Rozzy

print(res.\_\_next\_\_()) # Blaze

why generators?

* 1. optimizing memory space
  2. result need not be constructed all at once
  3. only one defect: generators are slow

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

vals = [5, 2, 7, 1, 12, 9, 11]

res = sorted((x\*\*2 for x in vals), reverse=True)

print(res)

# [144, 121, 81, 49, 25, 4, 1]

# Confused why it returned a list rather than a generator

*# that is because sorted() converts everything to a list and sorts it*