1) . What is the difference between enclosing a list comprehension in square brackets and parentheses?

2) What is the relationship between generators and iterators?

3) What are the signs that a function is a generator function?

4) What is the purpose of a yield statement?

5) What is the relationship between map calls and list comprehensions? Make a comparison and contrast between the two.

ANSWERS:

**1.** A Generator Expression is doing basically the same thing as a List Comprehension does, but the GE does it lazily. The difference is quite similar to the difference between [range and xrange](https://code-maven.com/range-vs-xrange-in-python).

A List Comprehension, just like the plain range function, executes immediately and returns a list.

A Generator Expression, just like xrange returns and object that can be iterated over.

The comparision is not perfect though, because in an object returned by the generator expression, we cannot access an element by index.

The difference between the two kinds of expressions is that the List comprehension is enclosed in square brackets [] while the Generator expression is enclosed in plain parentheses ().

l = [n\*2 for n in range(1000)] # List comprehension

g = (n\*2 for n in range(1000)) # Generator expression

Type

The type of resulting values are list and generator respectively:

print(type(l)) # 'list'

print(type(g)) # 'generator'

Size in memory

The size of the objects is 9032 bytes (the list), and 80 bytes (the generator):

print(sys.getsizeof(l)) # 9032

print(sys.getsizeof(g)) # 80

Access by Index

We can access the elements of the list, but if we try to access the elements of the generator we get a TypeError:

print(l[4]) # 8

print(g[4]) # TypeError: 'generator' object has no attribute '\_\_getitem\_\_'

Loop over

Finally, but most importantly, we can iterate over either of them:

for v in l:

pass

for v in g:

pass

The full example

**examples/python/generator\_expression.py**

1. #!/usr/bin/env python
2. from \_\_future\_\_ import print\_function
3. import sys
5. l = [n\*2 for n in range(1000)] # List comprehension
6. g = (n\*2 for n in range(1000)) # Generator expression
8. print(type(l)) # <type 'list'>
9. print(type(g)) # <type 'generator'>
11. print(sys.getsizeof(l)) # 9032
12. print(sys.getsizeof(g)) # 80
14. print(l[4]) # 8
15. #print(g[4]) # TypeError: 'generator' object has no attribute '\_\_getitem\_\_'
17. for v in l:
18. pass
19. for v in g:
20. pass

**2.Table of difference between Iterator vs Generators**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Iterator | Generator | | --- | --- | | Class is used to implement an iterator | Function is used to implement a generator. | | Local Variables aren’t used here. | All the local variables before the yield function are stored. | | Iterators are used mostly to iterate or convert other objects to an iterator using iter() function. | Generators are mostly used in loops to generate an iterator by returning all the values in the loop without affecting the iteration of the loop | | Iterator uses iter() and next() functions | Generator uses yield keyword | | Every iterator is not a generator | Every generator is an iterator | |

**3**.Python provides a generator to create your own [iterator function](https://www.tutorialsteacher.com/python/iter-method). A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values. In a generator function, a yield statement is used rather than a return statement. The following is a simple generator function.

def mygenerator():

print('First item')

yield 10

print('Second item')

yield 20

print('Last item')

yield 30

In the above example, the mygenerator() function is a generator function. It uses yield instead of return keyword. So, this will return the value against the yield keyword each time it is called. However, you need to create an iterator for this function, as shown below.

>>> gen = mygenerator()

>>> next(gen)

First item

10

>>> next(gen)

Second item

20

>>> next(gen)

Last item

30

The generator function cannot include the return keyword. If you include it, then it will terminate the function. The difference between yield and return is that yield returns a value and pauses the execution while maintaining the internal states, whereas the return statement returns a value and terminates the execution of the function.

The following generator function includes the return keyword.

def mygenerator():

print('First item')

yield 10

return

print('Second item')

yield 20

print('Last item')

yield 30

Now, execute the above function as shown below.

>>> gen = mygenerator()

>>> next(gen)

First item

10

>>> next(gen)

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

it.\_\_next\_\_()

StopIteration

As you can see, the above generator stops executing after getting the first item because the return keyword is used after yielding the first item.

## Using for Loop with Generator Function

The generator function can also use the for loop.

Example: Use For Loop with Generator Function

 Copy

def get\_sequence\_upto(x):

for i in range(x):

yield i

As you can see above, the get\_sequence\_upto function uses the yield keyword. The generator is called just like a normal function. However, its execution is paused on encountering the yield keyword. This sends the first value of the iterator stream to the calling environment. However, local variables and their states are saved internally.

The above generator function get\_sequence\_upto() can be called as below.

Example: Calling Generator Function

 Copy

>>> seq = get\_sequence\_upto(5)

>>> next(seq)

0

>>> next(seq)

1

>>> next(seq)

2

>>> next(seq)

3

>>> next(seq)

4

>>> next(seq)

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

it.\_\_next\_\_()

StopIteration

The function resumes when [next()](https://www.tutorialsteacher.com/python/next-method) is issued to the iterator object. The function finally terminates when next() encounters the StopIteration error.

In the following example, function square\_of\_sequence() acts as a generator. It yields the square of a number successively on every call of [next()](https://www.tutorialsteacher.com/python/next-method).

Example:

 Copy

def square\_of\_sequence(x):

for i in range(x):

yield i\*i

The following script shows how to call the above generator function.

gen=square\_of\_sequence(5)

while True:

try:

print ("Received on next(): ", next(gen))

except StopIteration:

break

The above script uses the try..except block to handle the StopIteration error. It will break the while loop once it catches the StopIteration error.

Output

Received on next(): 0

Received on next(): 1

Received on next(): 4

Received on next(): 9

Received on next(): 16

We can use the for loop to traverse the elements over the generator. In this case, the next() function is called implicitly and the StopIteration is also automatically taken care of.

Example: Generator with the For Loop

 Copy

squres = square\_of\_sequence(5)

for sqr in squres:

print(sqr)

Output

0

1

4

9

16

One of the advantages of the generator over the iterator is that elements are generated dynamically. Since the next item is generated only after the first is consumed, it is more memory efficient than the iterator.

## Generator Expression

Python also provides a generator expression, which is a shorter way of defining simple generator functions. The generator expression is an anonymous generator function. The following is a generator expression for the square\_of\_sequence() function.

Example: Generator Expression

 Copy

>>> squres = (x\*x for x in range(5))

>>> print(next(squre))

0

>>> print(next(squre))

1

>>> print(next(squre))

4

>>> print(next(squre))

9

>>> print(next(squre))

16

In the above example, (x\*x for x in range(5)) is a generator expression. The first part of an expression is the yield value and the second part is the for loop with the collection.

The generator expression can also be passed in a function. It should be passed without parentheses, as shown below.

Example: Passing Generator Function

 Copy

>>> import math

>>> sum(x\*x for x in range(5))

30

In the above example, a generator expression is passed without parentheses into the built-in function sum.

4. The heart of a generator function is the **yield** keyword. In its simplest form, a yield statement looks much like a return statement, except that instead of stopping execution of the function and returning, yield instead provides a value to the code looping over the generator and pauses execution of the generator function.

5.**Map VS List Comprehension**

* List comprehension is more concise and easier to read as compared to map.
* List comprehension allows filtering. In map, we have no such facility. For example, to print all even numbers in range of 100, we can write [n for n in range(100) if n%2 == 0]. There is no alternate for it in map
* List comprehension are used when a list of results is required as map only returns a map object and does not return any list.
* List comprehension is faster than map when we need to evaluate expressions that are too long or complicated to express
* Map is faster in case of calling an already defined function (as no lambda is required).

### Comparing Execution Time

Now to examine the execution performance of list comprehension and map function, we will import a module "timeit" to check the execution time. Let us see the result with a variety of methods.

Without lambda: Map is faster than List Comprehension when function is already defined in case of map function.

**Example code 1: This code will print the time taken to evaluate numbers from 1 to 50. Map function is used without lambda.**

import timeit

# list comprehension

l1 = timeit.timeit( '[ l for l in range(50)]' , number = 999999)

print (l)

#map function

f= 'def num( ) : print (n)'

m1 = timeit.timeit( ' map (num, range(50))' , number = 999999, setup = f )

print (m)

**With lambda in map:** List comprehension is better than map function when we don't define the function beforehand and use lambda expression inside map.

**Example code 2:** This code will print the time taken to add a number to itself and this is applied for each element of the list. The expression is defined as lambda expression.

import timeit

# list comprehension

l2 = timeit.timeit( '[ n+n for n in range(50)]' , number = 999999)

print (l)

#map function

m2 = timeit.timeit( ' map (lambda a: a+a, range(50))' , number = 999999, setup = f )

print (m)

Write the above codes on your IDE. You will see that for the first case, m1 is very less than l1. That means the map works faster than list comprehension. For the second case, m2 is greater than l2, implying that list comprehension is faster than map function when map function is used with Lambda expression.

**Conclusion**

There are no clear answers about which is the better option, in Python Map Vs List comprehension. We should know what the problem statement is, then use the method which suits us better because our main objective is to calculate the solution with optimal time and space consumption.

POOJA PATRA

BANGALORE