

## Key Term

**List Comprehension** - A concise syntax for creating lists that applies a function or operation to elements of an iterable

**Generator** - A function that returns an iterator object which yields one item at a time instead of returning a whole list

**yield** - A keyword used in generator functions to return a value from the function while retaining state

**Iterable** - An object that can return its members one at a time, allowing it to be iterated over in a loop

**Iterator** - An object that represents a stream of data that can be iterated over

```
1  # List comprehension with for loop to cube numbers
2  nums = [1, 2, 3, 4]
3  cubes = [num**3 for num in nums]
4  print(cubes) # [1, 8, 27, 64]
5
6  # Generator function yields numbers one by one
7  def num_sequence(n):
8      |   for i in range(n):
9      |       yield i
10
11  seq = num_sequence(5)
12  print(next(seq)) # 0
13  print(next(seq)) # 1
14
15  # Iterator from generator allows iteration
16  iterator = iter(num_sequence(3))
17  print(next(iterator)) # 0
18  print(next(iterator)) # 1
19
20  # Strings are iterable
21  chars = ["c" for c in "hello"]
22  print(chars) # ['h', 'e', 'l', 'l', 'o']
```

```
[1, 8, 27, 64]
0
1
0
1
['c', 'c', 'c', 'c', 'c']
```

# List Comprehensions in Python

## List Comprehensions

```
[10]: output = []
      for x in range(10):
          output.append(x**2)
```

0-9  
iterate through a sequence  
a new list  
return to

```
[10]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
[11]: output = [x**2 for x in range(10)]
```

1. first thing to put is the item to be appended to the output  
2. simulate the same syntax of the for-loop  
enclosed in square brackets

```
[11]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

3. assigning the comprehension to variable

- are a syntactic construct to perform the same function as a subset of for-loops.

replace with a list comprehension

- The list comprehension is a syntactic way, in more simple way, represent a for-loop that returns a list

## Map

```
[12]: import random
      scores = []
      for i in range(5):
          scores.append(random.randint(i, 10))
```

0-4  
then append to the list random  
5 times  
0-4  
sequence: range object  
function: random int function

```
[12]: [2, 8, 8, 5, 6]
```

```
[14]: [random.randint(i, 10) for i in range(5)]
```

comprehensions

```
[14]: [8, 9, 9, 3, 5]
```

## Filter

```
[15]: caps = []
      for letter in "Henry Honey":
          if letter.isupper():
              caps.append(letter)
```

then append  
if letter is upper case

```
[15]: ['H', 'H']
```

1. variable that we wish to append  
2. for loop  
3. filter/condition  
sequences

```
[15]: [letter for letter in "Henry Honey" if letter.isupper()]
```

## Nest

```
[17]: list_of_lists = [['a','b','c'], ['d','e','f'], ['g','h','i']]
      flat = []
      for sub_list in list_of_lists:
          for item in sub_list:
              flat.append(item)
```

list of lists  
we want to flatten  
so we do with nested for loop

```
[17]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

```
[18]: [item for sublist in list_of_lists for item in sublist]
```

variable to return  
get each list  
get each letter

```
[18]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

## Comprehensions advantage:

- Simple loops that are easier to read in 1 line
- avoid making overly complex list



# Generator Expressions in Python

- are similarly, memory efficient alternative to list comprehensions

## Generator Expressions

```
[1]: large_num = 9999999
    l_squares = [x**2 for x in range(large_num)]
```

list comprehension  
square bracket

```
[13]: import sys
    sys.getsizeof(l_squares)
```

```
[13]: 89095160
```

```
[15]: g_squares = (x**2 for x in range(large_num))
    sys.getsizeof(g_squares)
```

parentheses

```
[15]: 104
```

reduce memory

```
[16]: g_squares
```

appear only object not value

```
[16]: <generator object <genexpr> at 0x13d5ed000>
```

```
[18]: next(g_squares)
```

to access a value and a generator expression

```
[18]: 1
```

we can call it repeatedly the output will be next value

```
[19]: for x in g_squares:
    print(x)
    if x > 12:
        break
```

write a for loop to use this generator expression

because g expression will go a long way

4  
9  
16

## Chaining Generator Expressions

```
[22]: evens = (x for x in range(0, 100, 2))
    div_three = (y for y in evens if y%3 == 0)
    next(div_three)
```

2 generators that chained together  
iterates through the first generated expression

```
[22]: 0
```

```
[23]: [x for x in div_three]
```

to use

```
[23]: [6, 12, 18, 24, 30, 36, 42, 48, 54, 60, 66, 72, 78, 84, 90, 96]
```

# Generator Functions in Python

- For using more complicated Generator with its function

simple function

## Generator Functions

```
[10]: def return_num():
      for x in range(5):
          return x
      return_num()
[10]: 0
```

→ means that it will stop the first time it goes through the first iteration of the loop

← stop at the first iteration

## Generator Functions

```
[11]: def return_num():
      for x in range(5):
          yield x
      return_num()
[11]: <generator object return_num at 0x11710a260>
```

← from return to yield

→ return as object

## Generator Functions

```
[12]: def return_num():
      for x in range(5):
          yield x
      gen_num = return_num()
[13]: next(gen_num)
[13]: 0
```

→ it goes next every time when we executed it doesn't reset the loop

create a counting generator

```
[16]: def counter():
      x = 0
      while True:
          yield x
          x += 1
      count = counter()
[17]: next(count)
[17]: 0
```

← initiates a value to zero

← infinite for-loop

→ each iteration it yields a number and increments it up

→ every time, increment one

we can set an argument

```
[20]: def counter(x):
      while True:
          yield x
          x += 1
      count = counter(12)
[21]: next(count)
[21]: 12
```

← start as twelve, if we executed again, it'll be 13

create this with generator



## FIBONACCI SERIES

Default

0 1 1 2 3 5 8 13

0 + 1 = 1

1 + 1 = 2

1 + 2 = 3

2 + 3 = 5

3 + 5 = 8

5 + 8 = 13

last last

```
[23]: def fib():
      for cur in (0,1):
          last = cur
          yield cur
      while True:
          yield cur
          last, cur = cur, last + cur
[24]: f = fib()
[29]: next(f)
[29]: 0
```

you have to execute 2 times

first for 0 and 1

first iteration: 0

second iteration: 1

it will yield 2 times for 0 then 1 and 2

one iterating through the tuple

→ infinite-loop then reset variable values

finish the yield then go to infinite loop

first iteration

second iteration

- it powerful with small memory that need infinite loop