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

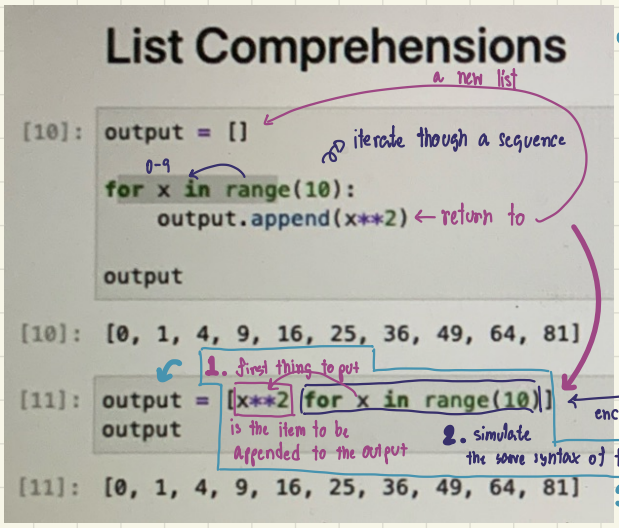
Iterator - An object that represents a stream of data that can be iterated over

```python
# List comprehension with for loop to cube numbers
nums = [1, 2, 3, 4]
cubes = [num**3 for num in nums]
print(cubes) # [1, 8, 27, 64]

# Generator function yields numbers one by one
def num_sequence(n):
    for i in range(n):
        yield i

seq = num_sequence(5)
print(next(seq)) # 0
print(next(seq)) # 1

# Iterator from generator allows iteration
iterator = iter(num_sequence(3))
print(next(iterator)) # 0
print(next(iterator)) # 1

# Strings are iterable
chars = ["c" for c in "hello"]
print(chars) # ['h', 'e', 'l', 'l', 'o']
```

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

List Comprehensions in Python

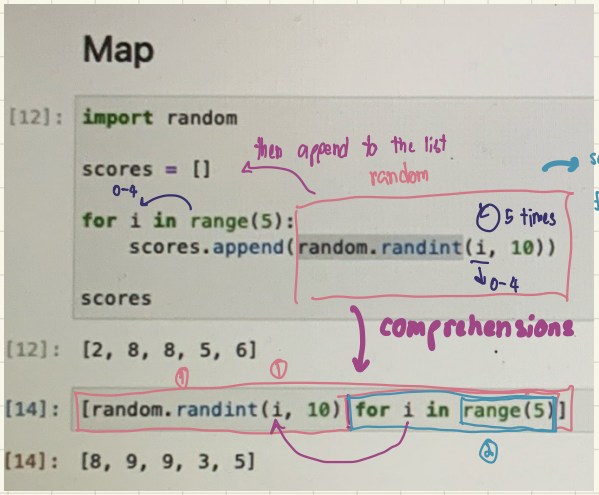
List Comprehensions



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

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

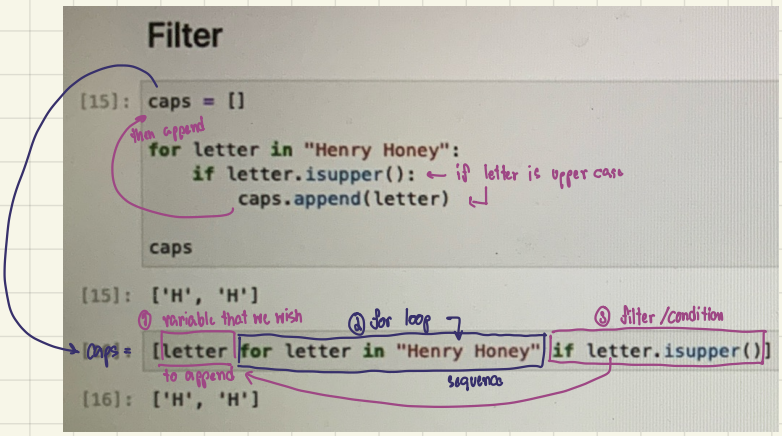
Map



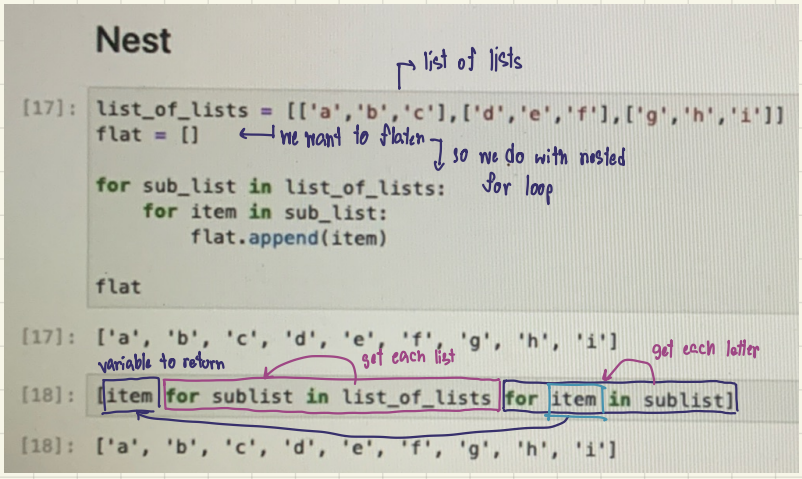
sequence : range object

function : random int function

Filter



Nest

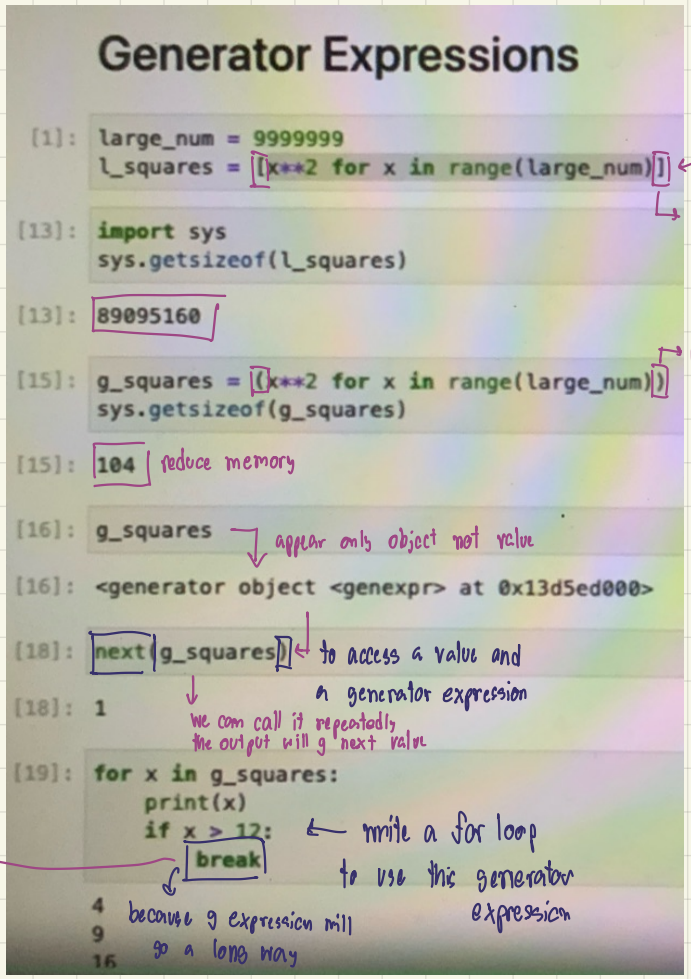


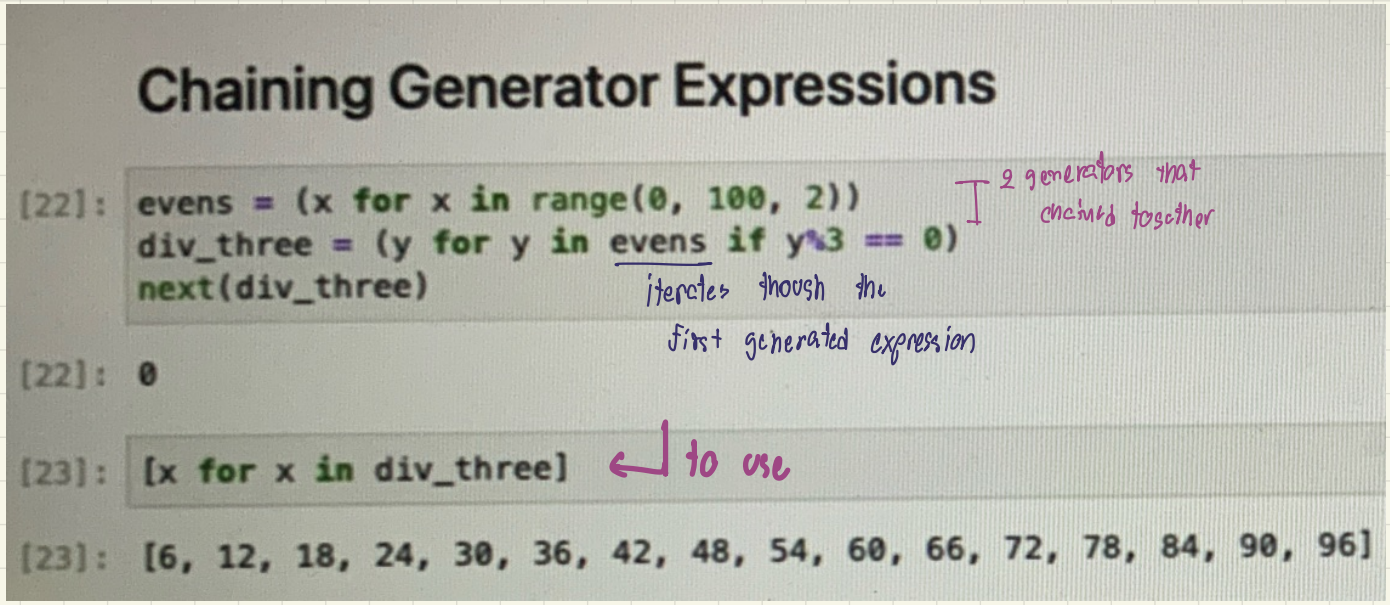
Comprehensions advantage:

- 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 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()
```

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

```
[10]: 0
```

← stop at the first iteration

Generator Functions

```
[11]: def return_num():
          for x in range(5):
              yield x
      return_num()
```

← from return to yield

↓ return as object

```
[11]: <generator object return_num at 0x11710a260>
```

Generator Functions

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

grab this

```
[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, id 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 (cur)

1 + 1 = 2 (cur)

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]: 3
```

you have to execute 2 times first for 0 and 1 — one iterating through the tuple

first iteration: 0 (last 0, cur 0)

second : 1 (1)

it will yield 2 times for 0 — then 1 and 2

finish the yield then go to infinite loop

first iteration

second iteration

0 1 1 2

↳ infinite-loop ↓ then reset variable values

- it powerful with small memory that need infinite loop