## Vòng lặp for trong python là gì?

Đối với vòng lặp for trong Python, giống như bất kỳ ngôn ngữ nào khác, được sử dụng để lặp lại một khối lệnh trong một số lần cố định. Mặc dù các ngôn ngữ khác chứa các lệnh điều kiện và biểu thức gia tăng. Nhưng trong Python, giá trị lặp và giá trị tăng được kiểm soát bằng cách tạo một chuỗi.

Đối với vòng lặp for trong Python được sử dụng cho các lần lặp tuần tự trong một số lần nhất định, nghĩa là theo độ dài của chuỗi. Lặp lại các chuỗi trong Python được gọi là traverse.

Cú pháp

**for**variable\_1, variable\_2, .. variable\_n in sequence:

    # for-block

Sequence ở đây là một **iterable object**(có thể là iterator hoặc là một dạng object cho phép sử dụng indexing hoặc thậm chí không phải hai kiểu trên).

**Lưu ý:** Nếu sequence là một **iterator object**thì việc dùng vòng lặp duyệt qua cũng sẽ tương tự như bạn sử dụng hàm next.

## Sử dụng vòng lặp để xử lí các iterator và Dict

>>> iter\_ = (x for x in range(3))

>>> iter\_ = (x for x in range(3))

>>> for value in iter\_:

... print('->', value)

...

-> 0

-> 1

-> 2

>>> value # biến value gián tiếp được khai báo

2

>>> next(iter\_) #. Đây là object chỉ dùng một lần.

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

Tiếp đến chúng ta sẽ dùng vòng lặp này để duyệt một Dict. Nếu như một số ngôn ngữ khác phải có một vòng lặp mới for-reach thì với Python lại không cần. **Dict-items** không phải là một **iterator object**. Cũng không phải là một object cho phép bạn indexing. Nhưng nó vẫn là một **iterable**, nên ta có thể dùng một constructor nào đó để biến đổi nó về một thứ dễ xem xét hơn.

>>> howkteam = {'name': 'Kteam', 'kter': 69}

>>> howkteam.items()

dict\_items([('name', 'Kteam'), ('kter', 69)])

>>> list\_values = list(team.items())

>>> list\_values

[('name', 'Kteam'), ('kter', 69)]

>>> list\_values[0]

('name', 'Kteam')

>>> list\_values[-1]

('kter', 69)

>>> for key, value in team.items():

... print(key, '=>', value)

...

name => Kteam

kter => 69

### **Vòng lặp for với string**

Mình thấy đây là một trong những ứng dụng rất hay của for trong python. Đối với các chuỗi, chúng ta có thể in ra các  kí tự tạo nên chuỗi đó bằng vòng lặp for:

## Generator trong Python

Để xây dựng một iterator, ta cần phải theo dõi khá nhiều bước ví dụ như: triển khai class với phương thức \_\_iter\_\_() và \_\_next\_\_(), theo dõi các tình trạng bên trong, StopIteration xảy ra khi không có giá trị nào được trả về...

Generator được dùng để giải quyết các vấn đề này. Generator là cách đơn giản để tạo ra iterator.

Nói một cách đơn giản, **generator là một hàm trả về một đối tượng (iterator) mà chúng ta có thể lặp lại (một giá trị tại một thời điểm).** Chúng cũng tạo ra một đối tượng kiểu danh sách, nhưng bạn chỉ có thể duyệt qua các phần tử của generator một lần duy nhất vì generator không lưu dữ liệu trong bộ nhớ mà cứ mỗi lần lặp thì chúng sẽ tạo phần tử tiếp theo trong dãy và trả về phần tử đó.

## Làm thể nào để tạo Generator trong Python

**Để tạo generator trong Python,** bạn sử dụng từ khóa def giống như khi định nghĩa một hàm. Trong generator, dùng câu lệnh yield để trả về các phần tử thay vì câu lệnh return như bình thường.

Nếu một hàm chứa ít nhất một yield (có thể có nhiều yield và thêm cả return) thì chắc chắn đây là một hàm generator. Trong trường hợp này, cả yield và return sẽ trả về các giá trị từ hàm.

Điều khác biệt ở đây là return sẽ chấm dứt hoàn toàn một hàm, còn yield sẽ chỉ tạm dừng các trạng thái bên trong hàm và sau đó vẫn có thể tiếp tục khi được gọi trong các lần sau.

Ví dụ, khi bạn gọi phương thức \_\_next\_\_() lần thứ nhất, generator thực hiện các công việc tính toán giá trị rồi gặp từ khóa yield nào thì nó sẽ trả về các phần tử tại ví trí đó, khi bạn gọi phương thức \_\_next\_\_() lần thứ hai thì generator không bắt đầu chạy tại vị trí đầu tiên mà bắt đầu ngay phía sau từ khóa yield thứ nhất. Cứ như thế generator tạo ra các phần tử trong dãy, cho đến khi không còn gặp từ khóa yield nào nữa thì giải phóng exception StopIteration.

## Sự khác biệt của hàm generator và hàm thông thường

Đây là một số khác biệt giữa hàm generator và [hàm thông thường](https://quantrimang.com/cac-ham-trong-python-143771):

* Hàm generator chứa một hoặc nhiều câu lệnh yield.
* Khi được gọi, generator trả về một đối tượng (iterator) nhưng không bắt đầu thực thi ngay lập tức.
* Các phương thức như \_\_iter \_\_() và \_\_next\_\_() được triển khai tự động. Vì vậy, chúng ta có thể lặp qua các mục bằng cách sử dụng next().
* Yield sẽ tạm dừng hàm, các biến cục bộ và trạng thái của chúng được ghi nhớ giữa các lệnh gọi liên tiếp. Mỗi lần lệnh yield được chạy, nó sẽ sinh ra một giá trị mới.
* Cuối cùng, khi hàm kết thúc, StopIteration sẽ xảy ra nếu tiếp tục gọi hàm.

Dưới đây là một ví dụ để minh họa tất cả các điểm đã nêu ở trên. Chúng ta có một hàm generator có tên my\_gen() với một số câu lệnh yield.

# Hàm generator đơn giản

def my\_gen():

n = 1

print('Doan text nay duoc in dau tien')

# Hàm Generator chứa các câu lệnh yield

yield n

n += 1

print('Doan text nay duoc in thu hai')

yield n

n += 1

print('Doan text nay duoc in cuoi cung')

yield n

>>> # Trả về một đối tượng nhưng không bắt đầu thực thi ngay lập tức.

>>> a = my\_gen()

>>> # Chúng ta có thể lặp qua các mục bằng cách sử dụng next().

>>> next(a)

Doan text nay duoc in dau tien

1

>>> # Yield sẽ tạm dừng hàm, quyền điều khiển chuyển đến người gọi

>>> # Các biến cục bộ và trạng thái của chúng được ghi nhớ giữa các

lệnh gọi liên tiếp.

>>> next(a)

Doan text nay duoc in thu hai

2

>>> next(a)

Doan text nay duoc in cuoi cung

3

>>> # Cuối cùng, khi hàm kết thúc, StopIteration sẽ xảy ra nếu tiếp

tục gọi hàm.

>>> next(a)

Traceback (most recent call last):

...

StopIteration

>>> next(a)

Traceback (most recent call last):

...

StopIteration

Có thể thấy trong ví dụ trên là giá trị của biến n được ghi nhớ giữa các lần gọi, không giống như các hàm bình thường kết thúc ngay sau mỗi lần gọi.

Khi bạn gọi phương thức next() lần thứ nhất, generator thực hiện các công việc tính toán giá trị rồi trả về phần tử tại ví trí đó, khi gọi phương thức next() lần thứ hai thì generator không bắt đầu chạy tại vị trí đầu tiên mà bắt đầu ngay phía sau từ khóa yield thứ nhất. Cứ như thế generator tạo ra các phần tử trong dãy, cho đến khi không còn gặp từ khóa yield nào nữa thì giải phóng exception StopIteration.

Để khởi động lại quá trình, tạo một đối tượng generator khác bằng cách sử dụng đối tượng như a = my\_gen().

**Lưu ý**: Có thể sử dụng generator trực tiếp cho các [vòng lặp for](https://quantrimang.com/vong-lap-for-trong-python-142234).

Vòng lặp for lấy một iterator và lặp lại nó bằng hàm next(), tự động kết thúc khi StopIteration xảy ra.

# Hàm generator đơn giản

def my\_gen():

n = 1

print('Doan text nay duoc in dau tien')

# Hàm Generator chứa câu lệnh yield

yield n

n += 1

print('Doan text nay duoc in thu hai')

yield n

n += 1

print('Doan text nay duoc in cuoi cung')

yield n

# Sử dụng vòng lặp for

for item in my\_gen():

print(item)

Chạy chương trình, kết quả trả về là

Doan text nay duoc in dau tien

1

Doan text nay duoc in thu hai

2

Doan text nay duoc in cuoi cung

3

**List Comprehension trong Python là gì?**

List Comprehension (cách viết code ngắn gọn để tạo một danh sách phức tạp) nghe có vẻ phức tạp nhưng thực tế không phải như vậy. Trong Python, nó chỉ đơn giản là cách nhanh chóng để lọc hoặc chỉnh sửa một danh sách dựa trên một số tiêu chí. List Comprehension giúp bạn viết code ngắn gọn (đặc biệt khi đang ở trong một vòng lặp đã có) và dễ đọc, nhìn code hơn.

Xem thêm: [Các kỹ thuật vòng lặp trong Python](https://quantrimang.com/cac-ky-thuat-vong-lap-trong-python-143741)

**Cách sử dụng List Comprehensions trong Python**

**Lưu ý**: Các ví dụ này đều sử dụng Python 3.6.

Hãy xem xét đoạn code này, sao chép một array (mảng) và biến mỗi chữ cái trong array này thành chữ viết hoa. Nó thực hiện điều này bằng cách lặp lại từng mục trong array.

letters = ['a', 'b', 'c', 'd']

print(letters)

upper\_letters = []

for letter in letters:

result = letter.upper()

upper\_letters.append(result)

print(upper\_letters

Bây giờ ví dụ dưới đây cũng tương tự như trên, nhưng thực hiện trong một dòng code duy nhất sử dụng List Comprehension Python cơ bản

letters = ['a', 'b', 'c', 'd']

print(letters)

upper\_letters = [x.upper() for x in letters]

print(upper\_letters)

Ví dụ này tạo ra một list (danh sách) được gọi là **letters** để lưu trữ các chữ thường “a”, “b”, “c” và “d”. Giả sử bạn muốn biến tất cả các phần tử trong danh sách này thành chữ hoa mà không sử dụng List Comprehension, bạn sẽ phải tạo một danh sách mới lưu trữ kết quả (được gọi là **upper\_letters**), lặp qua mọi phần tử trong danh sách **letters**, chuyển đổi từng chữ cái (và lưu trữ nó trong **result**), và sau đó thêm chữ viết hoa vào trong danh sách mới. Bạn thấy đấy, có quá nhiều việc cần làm.

List Comprehension ở đây gần như tương đương, thay thế vòng lặp. Nó chuyển đổi mỗi một chữ cái trong danh sách **letters** và trả về kết quả như một danh sách mới. List Comprehension chỉ làm việc trên danh sách và trả lại trên danh sách mới.

Có ba phần trong List Comprehension. List Comprehension phải bắt đầu và kết thúc bằng dấu ngoặc vuông (**[** và **]**). Đây là cách nó được thiết kế và cho phép Python biết rằng bạn sẽ làm việc với một danh sách.

**Iteration**là một khái niệm chung cho việc lấy từng phần tử một của một đối tượng nào đó, bất cứ khi nào bạn sử dụng vòng lặp hay kĩ thuật nào đó để có được giá trị một nhóm phần tử thì đó chính là Iteration.

**Ví dụ:** như bạn ăn một snack, bạn sẽ lấy từng miếng trong bọc snack ra ăn cho tới khi hết thì thôi. Bạn có thể coi việc lấy bánh là một vòng lặp. Đương nhiên bạn cũng có thể chọn không lấy hết số bánh ra.

**Iterable object**là một object có phương thức \_\_iter\_\_ trả về một iterator, hoặc là một object có phương thức \_\_getitem\_\_ cho phép bạn lấy bất cứ phần tử nào của nó bằng indexing ví dụ như Chuỗi, List, Tuple.

**Iterator object** đơn giản chỉ là một đối tượng mà cho phép ta lấy từng giá trị một của nó. Có nghĩa là bạn không thể lấy bất kì giá trị nào như ta hay làm với List hay Chuỗi.

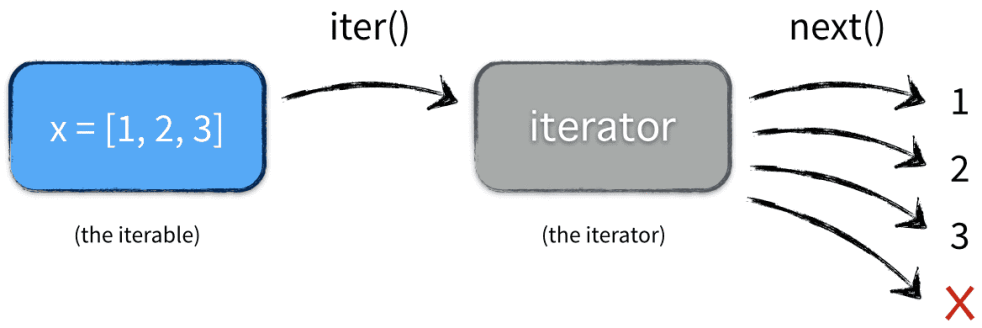
Iterator không có khả năng tái sử dụng trừ một số iterator có phương thức hỗ trợ như file object sẽ có phương thức seek.

Iterator sử dụng hàm next để lấy từng giá trị một. Và sẽ có lỗi **StopIteration**khi bạn sử dụng hàm next lên đối tượng đó trong khi nó hết giá trị đưa ra cho bạn.

Các **iterable object** chưa phải là iterator. Khi sử dụng hàm iter sẽ trả về một iterator. Đây cũng chính là cách các vòng lặp hoạt động.

Về mặt kỹ thuật, Iterator trong Python phải thực hiện hai phương thức đặc biệt là *\_\_iter\_\_()* và *\_\_next\_\_(),* gọi chung là giao thức iterator (Iterator Protocol)

* Phương thức *\_\_iter\_\_* trả về chính đối tượng iterator. Phương thức này được yêu cầu cài đặt cho cả đối tượng "iterable" và iterator để có thể sử dụng các câu lệnh for và in.
* Phương thức *\_\_next\_\_* trả về phần tử tiếp theo. Nếu không còn phần tử nào nữa thì sẽ có lỗi StopIteration xảy ra.



**Iter()**là một hàm dựng sẵn trong Python nhận đầu vào là một đối tượng iterable và trả về kết quả là một iterator.

# Khai bao mot list  
my\_list = [4, 7, 0, 3]  
   
# lay mot iterator bang cach su dung iter()  
my\_iter = iter(my\_list)  
   
## su dung next()  
   
#prints 4  
print(next(my\_iter))  
   
#prints 7  
print(next(my\_iter))  
   
## next(obj) chinh la obj.\_\_next\_\_()  
   
#prints 0  
print(my\_iter.\_\_next\_\_())  
   
#prints 3  
print(my\_iter.\_\_next\_\_())  
   
## Xay ra loi StopIteration vi het gia tri tra ve  
next(my\_iter)

Chạy chương trình, kết quả trả về là:

4  
7  
0  
3  
   
Traceback (most recent call last):  
File "<stdin>", line 24, in <module>  
next(my\_iter)  
StopIteration

Một cách tương tự trả về kết quả này là ta sử dụng [vòng lặp *for*](https://quantrimang.com/vong-lap-for-trong-python-142234).

>>> for element in my\_list:  
... print(element)  
...  
4  
7  
0  
3

Như chúng ta thấy trong ví dụ trên, vòng lặp for có thể lặp lại tự động thông qua việc sử dụng danh sách.

Trong thực tế, vòng lặp for có thể lặp lại trên bất kỳ iterable nào. Chúng ta hãy xem kỹ hơn về cách vòng lặp for được thực hiện trong Python.

for element in iterable:  
# do something with element

Được thực hiện tương tự như:

# iter\_obj là một iterator object tạo từ iterable  
iter\_obj = iter(iterable)  
   
# vòng lặp  
while True:  
 try:  
 # sử dụng next  
 element = next(iter\_obj)  
 except StopIteration:  
 # nếu xảy ra lỗi StopIteration thì vòng lặp sẽ được break ra ngoài  
 break

Ở ví dụ này, bên trong vòng lặp for ta tạo một iterator object tên là iter\_obj bằng cách gọi iter() trên iterable.

Và như bạn thấy thì vòng lặp for ở đây chính là [vòng lặp while](https://quantrimang.com/vong-lap-while-trong-python-142407) vô hạn. Next() bên trong vòng lặp lấy ra các phần tử để thực hiện các câu lệnh ở for loop. Đến khi lấy hết các giá trị thì [ngoại lệ](https://quantrimang.com/error-va-exception-trong-python-160166) StopIteration sẽ được sinh ra và vòng lặp kết thúc.

Chúng ta có thể tự xây dựng iterator là một class. Xây dựng Iterator rất dễ dàng trong Python, chúng ta chỉ cần thực hiện các phương thức \_\_iter\_\_() và \_\_next\_\_().

class PowTwo:  
   
 def \_\_init\_\_(self, max = 0):  
 self.max = max  
   
 def \_\_iter\_\_(self):  
 self.n = 0  
 return self  
   
 def \_\_next\_\_(self):  
 if self.n <= self.max:  
 result = 2 \*\* self.n  
 self.n += 1  
 return result  
 else:  
raise StopIteration

Phương thức \_\_iter\_\_ sẽ làm đối tượng trở thành đối tượng iterable.

Giá trị trả về của \_\_iter\_\_ là một iterator. Nó cần có phương thức \_\_next\_\_ và trả về StopIteration nếu không còn phần thử nào nữa.

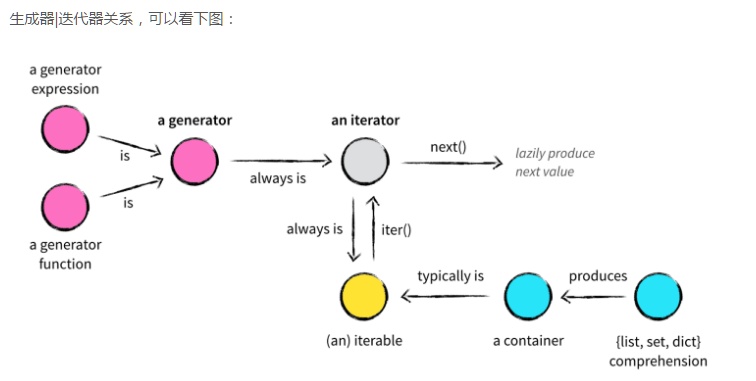
Tạo ra một iterator và chạy chương trình như sau:

>>> a = PowTwo(4)  
>>> i = iter(a)  
>>> next(i)  
1  
>>> next(i)  
2  
>>> next(i)  
4  
>>> next(i)  
8  
>>> next(i)  
16  
>>> next(i)  
Traceback (most recent call last):  
...  
StopIteration

Ưu điểm của việc sử dụng Iterator lặp là chúng tiết kiệm tài nguyên. Giống như hiển thị ở trên, bạn có thể nhận được tất cả các số lẻ mà không lưu trữ trên toàn bộ hệ thống số ở bộ nhớ.

**Comprehension** là một công cụ rất hiệu quả của Python để xử lí rất nhiều việc mà chỉ cần một dòng.

Bên cạnh đó. Người ta còn so sánh những **comprehension** và những đoạn code với chức năng tương tự thì comprehension có tốc độ nhanh hơn.



**(An) Iterable**

**A collection**

**{List, Set, Dict}  
Comprehension**

**MỐI QUAN HỆ GIỮA CÁC INTERTOOLS**

**is**

**is**

**always is**

**always is**

**inter()**

**Typically is**

**lazily produce next value**

**Produces**

**next()**

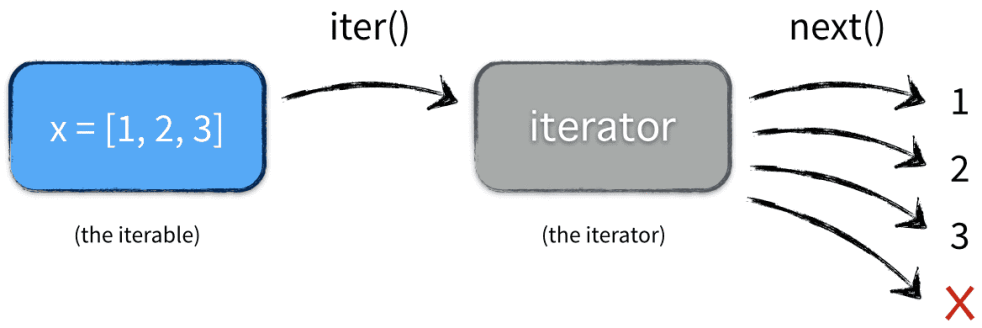
**An iterator**

**An Iterator**

**A Generator**

**A Generator funtion**

**A Generator funtion**





1

3

2

**X**

Iterator

THE ITERABLE

THE ITERATOR

Iter()

next()

Arr = [1, 2, 3]