Bài 33: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – YIELD

Xem bài học trên website để ủng hộ Kteam: Kiểu dữ liệu function trong Python – Yield

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage How Kteam nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – RETURN.

Và ở bài này Kteam sẽ lại tìm hiểu với các **KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – YIELD**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON.
- Xem qua bài CÁCH CHẠY CHƯƠNG TRÌNH PYTHON.
- Nắm CÁCH GHI CHÚ và BIẾN TRONG PYTHON.
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- CÂU ĐIỀU KIỆN IF TRONG PYTHON
- VÒNG LẶP WHILE và VÒNG LẶP FOR TRONG PYTHON

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Nhắc lại khái niệm iterables
- Giới thiệu generator
- Lệnh yield
- Phương thức send
- Vì sao nên dùng yield

Nhắc lại khái niệm iterables

Kteam đã từng giới thiệu với các bạn khái niệm này ở bài ITERATION & MỘT SỐ HÀM HỖ TRỢ CHO ITERATION OBJECT TRONG PYTHON. Và ở bài này, chúng ta sẽ nhắc lại vài khái niệm trước khi đi đến lệnh yield

Khi bạn tạo ra một list, bạn có thể truy xuất lần lượt từng giá trị của list đó. Người ta gọi đó là iteration

```
>>> kteam_lst = [1, 'Kteam', 2]
>>> for value in kteam_lst:
...     print(value)
...
1
Kteam
2
```

"**kteam_lst**" ở đây được gọi là một iterable. Mọi thứ mà bạn có thể dùng cú pháp "**for ... in ...**" đều là một **iterable**. Ví dụ như chuỗi, list, tuple, file,..

Nhưng iterable này rất thuận tiện cho chúng ta lưu dữ và truy xuất thông tin. Và để được như vậy bạn phải lưu trữ những thông tin đó trong các vùng nhớ máy tính của bạn. Vì lẽ đó, sẽ có trường hợp bạn không cần thiết phải giữ tất cả thông tin cùng một lúc vì nó quá nhiều.

Giới thiệu generator

Generator là iterator, một dạng của iterable nhưng khác ở chỗ bạn không thể tái sử dụng. Vì sao lại như vậy? **Generator** không lưu trữ tất cả các giá trị của bạn ở bộ nhớ, mà nó sinh ra lần lượt

```
>>> kteam_gen = (value for value in range(3))
>>> for value in  kteam_gen:
...     print(value)
...
0
1
2
```

Như đã nói, **generator** cũng là một iterable, nên nó cũng khá tương tự như khi bạn dùng **list** hoặc **tuple**. Nhưng, nếu bạn thử tái sử dụng generator đó

```
>>> for value in kteam_gen:
...     print(value)
...
>>>
```

Bạn thấy đấy, không có giá trị nào được in ra. Bởi vì khi nó sinh ra giá trị đầu tiên là 0, khi bạn kêu nó sinh tiếp giá trị 1, nó sẽ vứt bỏ giá trị 0 để nhường chỗ cho giá trị 1, và nếu bạn tiếp tục yêu cầu sinh thêm giá trị nó sẽ lại tiếp tục công việc như cũ cho tới khi kết thúc.

Lệnh yield

Các bạn chú ý: **y-i-e-l-d**, **yield**. Lệnh này khá là khó nhớ đặc biệt với những người chưa quen với tiếng Anh. Bạn cũng nên tra google để biết ý nghĩa của từ yield. Điều này sẽ giúp bạn biết rõ hơn lệnh này.

Lệnh này cách sử dụng gần giống với lệnh **return**, tuy nhiên nó khác **return** ở chỗ trả về một **object** thì **yield** sẽ trả về một generator.

Chúng ta hãy đến với một ví dụ với **return** sau đó ta sẽ so sánh nó với yield

```
>>> def square(lst):
...     sq_lst = []
...     for num in lst:
...             sq_lst.append(num**2)
...     return sq_lst
...
>>> kteam_ret = square([1, 2, 3])
>>> for value in kteam_ret:
...     print(value)
...
1
4
9
```

Và đây là khi sử dụng lệnh yield thay cho return

```
>>> def square(lst):
...     for num in lst:
...             yield num**2
...
>>> kteam_gen = square([1, 2, 3])
>>> for value in kteam_gen:
...     print(value)
...
1
4
9
```

Như bạn thấy, vì **return** sẽ quăng lại một list lưu trữ toàn bộ giá trị sau khi bình phương, thế nên bạn phải tạo một **list** để lưu hết những giá trị đó. Tuy nhiên, điều này là **không cần thiết** với **yield**. Nó sẽ lần lượt sinh ra từng giá trị bình phương một mà không cần một list để lưu trữ. Mỗi lần bạn gọi nó, nó sẽ chạy vào sinh ra cho bạn giá trị bạn cần như việc bạn sử dụng vòng lặp for để đọc từng giá trị trong một list.

Khi bạn dùng **yield** trong một hàm và khi gọi hàm đó, những dòng lệnh trong hàm sẽ không chạy ngay. Nó trả về một generator. Và mỗi khi bạn yêu cần nó sinh thì nó mới bắt đầu chạy vào bên trong thực hiện những dòng lệnh trong hàm **CHO TỚI KHI GẶP LỆNH YIELD** và nó sẽ sinh ra giá trị bạn yêu cầu yield, hàm bây giờ được tạm dừng. Bạn cần lưu ý, là chỉ tạm dừng, có nghĩa là nếu lần sau gọi, hàm sẽ tiếp tục chạy ở phần đó không phải chạy lại từ đầu

Khi nào thì **yield** hết? Khi mà nó đi hết phần còn lại của hàm mà không gặp lệnh **yield**.

Bạn sẽ hiểu rõ hơn khi xem hai ví dụ sau đây.

```
>>> def gen():
...     for value in range(3):
...             print('yield', value + 1, 'times')
...             yield value
...
>>> for value in gen():
...     print(value)
...
yield 1 times
0
yield 2 times
1
yield 3 times
2
```

```
>>> def gen():
...     yield 'Kteam'
...     print('this is the second yield')
...     yield 'Free education'
...     print('this is the last yield')
...     yield 'Long đẹp trai'
...     print('Will not return anything')
...
>>> for value in test():
...     print(value)
Kteam
this is the second yield
Free education
this is the last yield
Long đẹp trai
Will not return anything
```

Bạn cũng cần lưu ý thêm, nếu không có giá trị **yield** khi được gọi tiếp thì sẽ **yield** sẽ không trả về bất cứ thứ gì, có nghĩa là **None object** cũng không được trả về.

> **Lưu ý:** ngoài cách dùng **for** như bên trên để duyệt các generator, Kteam đã giới thiệu với các bạn hàm **next** ở bài ITERATION & MỘT SỐ HÀM HỖ TRỢ CHO ITERATION OBJECT TRONG PYTHON – một hàm để giúp bạn làm công việc tương tự.

Phương thức send

> **Lưu ý:** Bạn đọc cần đọc và ngẫm thật kĩ yield ở phía trên trước khi đọc đến phần này.

Đây là phương thức giúp bạn gửi giá trị vào trong một generator.

Cú pháp:

```
generator.send(value)
```

Bạn cũng không cần phải lo lắng nếu không hiểu được đoạn code dưới đây

```
>>> def gen():
...     for i in range(4):
...             x = yield i
...             print('value sent from you', x)
...
>>> g = gen() # gán generator này cho biến g
>>> next(g) # gọi hàm next để chạy lệnh yield "x = yield i"
0
>>> g.send('Kteam') # x vừa nãy khi gán cho biến yield giờ sẽ được gửi giá trị
value sent from you Kteam
1
>>> g.send('Free education')
value sent from you Free education
2
```

```
>>> next(g) # lần này ta không dùng send, mặc định giá trị gửi vào là None
value sent from you None
3
```

Đây là một ví dụ khác nữa về phương thức **send**. Một lần nữa, hãy coi thật kĩ ví dụ **send** vừa trên trước khi đến với ví dụ tiếp sau đây

```
>>> def gen():
...     while True:
...             x = yield # ở đây ta đang yield None, vì ta không cần thiết sinh giá trị gì ở đây
...             yield x ** 2
...
>>> g = gen()
>>> next(g) # chạy lệnh yield để ta gửi giá trị cho biến x lần sau
>>> g.send(2)
4
>>> next(g) # tiếp tục chạy yield để có thể gửi giá trị
>>> g.send(10)
100
```

Vì sao nên dùng yield

Tốc độ, khi sử dụng **generator**, để duyệt các giá trị thì generator sẽ nhanh hơn khi khi bạn duyệt một iterable lưu trữ một lúc tất cả các giá trị

Bộ nhớ, bạn sẽ phải cân nhắc việc dùng yield khi bạn làm việc với những tập dữ liệu lớn. Lúc đó, bạn sẽ phải xem xét lại xem liệu bạn có cần giữ tất cả các giá trị một lúc không hay chỉ cần sinh ra từng giá trị một để tiết kiệm bộ nhớ.

Còn một số ưu điểm nữa của yield, bạn đọc có thể tham khảo câu trả lời sau trên Stack Overflow:

> https://stackoverflow.com/questions/102535/what-can-you-use-python-generator-functions-for

---

Kết luận

Qua bài viết này, Bạn đã biết về lệnh yield trong hàm.

Ở bài tiếp theo, Kteam sẽ nói đến HÀM NẶC DANH TRONG PYTHON.

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quyên "**Luyện tập – Thử thách – Không ngại khó**".