Bài 35: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - FUNCTIONAL TOOLS

Xem bài học trên website để ủng hộ Kteam: Kiểu dữ liệu Function trong Python - Functional tools

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 – LAMBDA trong Python.

Và ở bài này Kteam sẽ lại tìm hiểu với các **KIỂU DỮ LIỆU FUNCTION – FUNCTIONAL TOOLS** trong Python.

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

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

- Hàm map
- Hàm filter
- Hàm reduce

Hàm map

Chúng ta thường hay phải xử lí các phần tử của một **list** hoặc một container nào đó bằng một phương thức.

Giả sử ta phải cập nhật một **list** bằng cách tăng mỗi giá trị trong **list** đó lên 1 đơn vị

```
>>> kteam = [1, 2, 3, 4]
>>> kteam_updated = []
>>> for value in kteam:
...     kteam_updated.append(value + 1)
...
>>> kteam_updated
[2, 3, 4, 5]
```

Chúng ta sẽ ngó sơ qua cú pháp của hàm **map** trước khi xem hàm map xử lí công việc bên trên. Đầu tiên là cái cơ bản nhất

> **map(func, iterable)**

Hàm map này sẽ trả về một **map object** (một dạng generator).

Vậy hàm **map** hoạt động như thế nào? Nôm na là hàm map lấy từng phần tử của iterable sau đó dùng gọi hàm **func** với **argument** là giá trị mới lấy ra từ iterable, kết quả trả về của hàm **func** sẽ được **yield**.

Nôm na hàm **map** nó sẽ trông như thế này:

```
>>> def mymap(func, iterable):
...     for x in iterable:
...             yield func(x)
```

Chúng ta đến với ví dụ để hiểu hơn

```
>>> def inc(x): return x + 1
...
>>> kteam = [1, 2, 3, 4]
>>> list(map(inc, kteam)) # dùng constructor list để ta dễ quan sát dữ liệu
[2, 3, 4, 5]
```

Bạn còn nhớ **lambda** chứ? (Nếu không có thể tham khảo bài HÀM NẶC DANH LAMBDA)

```
>>> kteam = [1, 2, 3, 4]
>>> list(map(lambda x: x + 1, kteam))
[2, 3, 4, 5]
```

Đôi lúc, việc sử dụng hàm map còn nhanh hơn cả **list comprehension**

```
>>> inc = lambda x: x + 1
>>> kteam = [1, 2, 3, 4]
>>>
>>> [inc(x) for x in kteam
[2, 3, 4, 5]
>>> list(map(inc, kteam))
[2, 3, 4, 5]
```

> **Lưu ý:** Ở **list comprehension** trên, nếu bạn thay ngoặc vuông (**[**) bằng ngoặc tròn (**(**) thì thời gian của ngoặc tròn có thể tương đương với hàm map và tiết kiệm dữ liệu hơn list comprehension vì nó cũng tạo ra một generator expression.

Ta mở rộng hàm map ra nhé. Vì đầy đủ cú pháp hàm map là

> **map(func, *iterable)**

Bạn đọc lưu ý, khi bạn pass vào nhiều **container** để biến hàm map gọm lại bằng cách packing argument thì các **container** phải cùng số lượng giá trị (cùng giá trị hàm len). Vì khi có nhiều container pass vào, thì hàm map sẽ cùng một lúc lấy lượt các giá trị của các **container**.

Bạn đọc sẽ hiểu kĩ khi xem ví dụ sau đây

```
>>> func = lambda x, y: x + y
>>>
>>> kteam_1 = [1, 2, 3, 4]
>>> kteam_2 = [5, 6, 7, 8]
>>>
>>> kteam = map(func, kteam_1, kteam_2)
>>> list(kteam)
[6, 8, 10, 12]
```

Như bạn thấy, hàm map sẽ lấy từng giá trị một của cả hai list rồi gửi nó vào hàm. À, bạn cũng phải lưu ý là bạn pass vào **n container** thì bạn cũng phải thiết kế cái hàm nào có thể nhận **n argument** luôn nhé.

```
>>> pow(2, 3) # 2^3
8
>>> pow(3, 4) # 3^4
81
>>> list(map(pow, [1, 2, 3], [2, 2, 2,])) # 1^2, 2^2, 3^2
[1, 4, 9]
```

Hàm filter

Filter có nghĩa là bộ phận lọc. Nghe qua, chắc bạn cũng ít nhiều biết được nó sẽ làm gì rồi.

Cú pháp hàm này như sau:

> **filter(function or None, iterable)**

Cũng như hàm map, hàm **filter** sẽ trả về một **filter object** (một dạng generator object)

> **Lưu ý:** không như hàm map, iterable ở đây chỉ là 1 container, không hề có packing argument.

Hàm **filter** lấy từng giá trị trong iterable, sau đó gửi vào hàm, nếu như giá trị hàm trả ra là một giá trị mà khi chuyển sang kiểu dữ liệu **boolean** là **True** thì sẽ yield giá trị đó, nếu không thì bỏ qua.

Trường hợp bạn không gửi hàm vào mà là **None**, hàm **filter** lấy từng giá trị trong iterable, nếu giá trị đó chuyển sang giá trị **boolean** là **True** thì yield, nếu không thì bỏ qua.

Chúng ta đến với ví dụ để hiểu thêm. Đầu tiên sẽ làm có function bằng một ví dụ lọc lấy các số dương (lớn hơn 0)

```
>>> func = lambda x: x > 0
>>>
>>> kteam = [1, -3, 5, 0, 2, 6, -4, -9]
>>> list(filter(func, kteam))
[1, 5, 2, 6]
```

Hàm **func** nhận vào 1 giá trị, nếu giá trị đó lớn hơn 0 thì trả về **True**, còn không thì là **False** nhờ toán tử so sánh. Vậy nên, các giá trị gửi vào mà nhận giá trị **False** là không được yield.

Nếu bạn nào còn mông lung thì bạn hãy xem qua **list comprehension** tương đương với hàm filter trên

```
>>> kteam = [1, -3, 5, 0, 2, 6, -4, -9]
>>> [x for x in kteam if x > 0]
[1, 5, 2, 6]
```

Tiếp đến là một ví dụ khác khi ta gửi **None** thay vì một function

```
>>> kteam = [0, None, 1, 'Kteam', '', 'Free Education', 69, False]
>>> list(filter(None, kteam))
[1, 'Kteam', 'Free Education', 69]
```

Hàm reduce

Bất cứ giá trị nào khi chuyển qua giá trị **boolean** mà **False** thì sẽ không được yield. Đơn giản phải không nào? :D

> **Lưu ý:** Với những bạn dùng Python 2.X, hàm reduce là hàm có sẵn, bạn chỉ việc dùng, còn với Python 3.X, nó đã được đưa vào trong thư viện functools. Vì thế nếu bạn muốn sử dụng nó, đừng quên import nhé.

```
from functools import reduce
```

Ở các ví dụ tiếp theo, thì sẽ không có dòng này vì lặp lại nên bạn đọc coi như chúng ta đã có dòng lệnh này ở đầu chương trình tức có nghĩa là chúng ta đã **import** hàm reduce từ thư viện functools rồi.

Hàm này kha khá phức tạp này, các bạn không cần nóng vội để hiểu nó. Ta hãy đến với cú pháp của nó.

```
reduce(function, sequence[, initial])
```

> **Lưu ý:** Hàm reduce không giống như hai hàm trước là trả về một **generator expression** mà là một giá trị.

Để đơn giản nhất, chúng ta hãy tạm chưa xét tới **parameter initial.**

Đầu tiên, hàm reduce sẽ lần lượt lấy hai giá trị đầu tiên của **sequence (index 0, index 1)** và đưa vào hàm function

> **Lưu ý:** đưa theo thứ tự (index 0, index 1)
>
> Hàm function này sẽ trả ra một giá trị (ta kí hiệu là A). Sau đó lấy tiếp giá trị thứ ba của sequence (index 2), rồi gửi vào function cũng theo thứ tự (A, index 2), rồi lại lặp lại như thế cho tới khi hết sequence.

Hãy đến với các ví dụ để hiểu hơn

Ví dụ dùng reduce để tính tổng các số trong list

```
>>> kteam_add = lambda x, y: x + y
>>> kteam = [1, 2, 3, 4, 5]
>>> reduce(kteam_add, kteam) # ((((1+2)+3)+4)+5)
15
```

Ví dụ dùng reduce để tính tích các số trong list

```
>>> kteam_multi = lambda x, y: x * y
>>> kteam = [1, 2, 3, 4]
>>> reduce(kteam_multi, kteam) # ((((1*2)*3)*4)
24
```

Nào, giờ chúng ta tới bước khi có **argument** cho **parameter initial**. Khi nãy, khi chưa có **initial**, hàm reduce lấy hai giá trị để quăng vào function đầu tiên. Nhưng khi bạn đưa argument vào cho parameter initial thì hàm reduce sẽ lấy giá trị initial và giá trị đầu tiên của **sequence (index 0)** đưa vào function và tiếp tục trả ra một giá trị, rồi giá trị đó lại tiếp tục với giá trị thứ hai của **sequence (index 1)**.

Ví dụ để hiểu thêm

```
>>> kteam = [1, 2, 3, 4]
>>> kteam_add = lambda x, y: x + y
>>> kteam_multi = lambda x, y: x * y
>>>
>>> reduce(kteam_add, kteam, 10)
20
>>> reduce(kteam_multi, kteam, 10)
240
```

Kết luận

Qua bài viết này, Bạn đã biết về hàm nặc danh lambda.

Ở bài tiếp theo, Kteam sẽ nói đến KỸ THUẬT ĐỆ QUY.

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**".