Bài 34: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – LAMBDA

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

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 YIELD – KIỂU DỮ LIỆU FUNCTION TRONG PYTHON.

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

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

- Mở đầu
- Giới thiệu lambda
- Vì sao dùng lambda?
- Câu điều kiện cho lambda
- Lambda chồng lambda

Mở đầu

Ngoài từ khóa "**def**", Python cũng hỗ trợ cho bạn một cách khác để có thể khai báo một function object, đó chính là **lambda**. Nó chỉ khác từ khóa "**def**" ở chỗ, thay vì **def** tạo một hàm với một cái tên xác định thì **lambda** trả về một hàm. Thế nên người ta hay gọi **lambda** là **hàm nặc danh (anonymous)**. Nó thường được sử dụng thường xuyên để có thể tạo ra một hàm chỉ với một dòng lệnh.

Giới thiệu lambda

Ta có cú pháp sau:

> **lambda argument_1, argument_2, ..., argument_n : expression**

Như đã nói ở trên, **lambda** hoạt động như khi bạn dùng từ khóa "**def**" khai báo hàm. Tuy nhiên, vẫn có một vài ưu điểm nổi trội của **lambda** so với cách bình thường:

- **lambda** là một expression, không phải là một câu lệnh. (Khái niệm expression đã được Kteam giới thiệu). Do đó **lambda** có thể có ở một vài chỗ mà "def" không thể có (bạn đọc sẽ biết ở phần sau)
- **lambda** là một dòng expression duy nhất, không phải là một khối lệnh. Phần expression của **lambda** giống với phần khối lệnh của hàm với một lệnh return ở cuối hàm nhưng với **lambda** bạn chỉ cần ghi giá trị mà

không cần ghi return. Bạn đọc sẽ hiểu rõ hơn ở phần sau khi biết **lambda** có thể sử dụng các câu lệnh điều kiện mà không cần phải sử dụng tới lệnh "**if**". Nhờ được thiết kế như vậy, **lambda** được ưu tiên dùng cho việc tạo ra những hàm đơn giản, còn nếu phức tạp thì ta sẽ sử dụng đến từ khóa "def".

Để có thể hiểu hơn, mời bạn đọc xem qua các ví dụ sau đây

Đây là khi bạn sử dụng từ khóa "def"

```
>>> def ave(a, b, c):
...     return (a + b + c)/3
...
>>> ave(1, 3, 2)
2.0
```

Còn đây là khi sử dụng lambda

```
>>> ave = lambda a, b, c: (a + b+ c)/3
>>> ave(1, 3, 2)
2.0
```

Bạn còn nhớ **default argument** chứ?

```
>>> def x_power_a(x, a = 2):
...     return x ** a
...
>>> x_power_a(2)
4
>>> x_power_a(2, 3)
8
```

Điều đó cũng có thể làm được với lambda

```
>>> x_power_a = lambda x, a=2: x ** a
>>> x_power_a(2)
4
>>> x_power_a(2, 3)
8
```

Bạn cũng lưu ý thêm là lambda cũng phân biệt **global** và **local** nhé.

```
>>> def kteam():
...     mem = lambda x: x + ' is a member of Kteam'
...     return mem # trả về một hàm nặc danh
...
>>> call_mem = kteam() # lấy biến call_mem giữ hàm nặc danh
>>> call_mem('Long') # giá  trị chuỗi được đưa vào cho biến x
'Long is a member of Kteam'
>>> call_mem('Giau')
'Giau is a member of Kteam'
>>> call_mem
<function kteam.<locals>.<lambda> at 0x03C2FDB0>
```

Vì sao dùng lambda?

Chung quy thì **lambda** là một công cụ nhanh gọn để bạn có thể tạo ra một hàm và sử dụng nó. Việc sử dụng nó thay cho "def" hay không là tùy ở bạn. Đương nhiên là bạn có thể chỉ sử dụng "def" thôi cũng được, hoàn toàn được, đặc biệt là những lúc mà hàm của bạn phức tạp, cần nhiều câu lệnh thì bạn không cần phải suy nghĩ nhiều nữa mà nên dùng "def" luôn. Nhưng giả sử bạn chỉ cần khởi tạo một hàm cấu trúc đơn giản và tái sử dụng nhiều lần thì sao? Lúc đó hãy nghĩ tới **lambda** nhé!

Chúng ta đến với một số ví dụ mà **lambda** hoàn toàn chiếm ưu thế so với "def".

```
>>> kteam_lst = [lambda x: x**2, lambda x: x**3, lambda x: x**4] # một list với các phần tử là các hàm nặc danh
>>> kteam_lst[0]
<function <lambda> at 0x002CC618>
>>> kteam_lst[0](2) # 2**2
4
>>> kteam_lst[-1](4) # 4**4
256
>>> for func in kteam_lst:
...     func(3) # 3**2, 3**3, 3**4
...
9
27
81
```

Rất tiện lợi phải không nào, dĩ nhiên điều này "def" không thể có vì như đề cập ở phần trên, **lambda** là một expression, không phải một câu lệnh. Nên **lambda** có thể ở nhiều nơi mà "def" không thể.

Với ví dụ bên trên khi bạn muốn sử dụng "def", bạn phải khởi tạo hàm ở ngoài rồi đưa vào **list**.

```
>>> def f1(x): return x**2
...
>>> def f2(x): return x**3
...
>>> def f3(x): return x**4
...
>>> kteam_lst = [f1, f2, f3]
>>> kteam_lst[0]
<function f1 at 0x00C8FE40>
>>> kteam_lst[-1](2) # 2**4
16
>>> for func in kteam_lst:
...     func(3)
...
9
27
81
```

Không chỉ mình **list**, bạn có thể sử dụng **lambda** với **dictionary**. Mời bạn đọc xem ví dụ sau đây:

```
>>> key = 'Kteam'
>>> {'Google': lambda: 'Goooooooog',
... 'YouTube': lambda: 'Youuuuuuuuu',
... 'Kteam': lambda: 'Free Education'}[key]()
'Free Education'
```

> **Lưu ý:** Bạn để ý ví dụ trên, phần **argument** của **lambda** ta để trống, điều này hoàn toàn đúng cú pháp vì phần **argument** là **optional** (không bắt buộc) nhưng phần expression bắt buộc phải có một expression.

Ta thử lại ví dụ trên nhưng không dùng **lambda** mà dùng "def"

```
>>> def f1(): return 'Gooooooooog'
...
>>> def f2(): return 'Youuuuuuuuu'
...
>>> def f3(): return 'Free Education'
...
>>> key = 'Kteam'
>>> {'Google': f1, 'YouTube': f2, 'Kteam': f3}[key]()
'Free Education'
```

Nó cũng như **lambda** thôi, nhưng rõ ràng ta thấy **lambda** tiện lợi hơn "def" dù cho chỉ vài dòng code. Điểm mạnh vượt trội của **lambda** so với "def" hoàn toàn được thể hiện với những hàm tính toán đơn giản nhanh chóng. Hơn thế nữa, khi dùng "def", bạn phải tạo ra một cái tên cho nó, và đôi khi việc bạn nghĩ ra một cái tên cho một cái hàm thực sự không hề đơn giản (việc này khá hiếm nhưng đã xảy ra).

Bạn sẽ còn thấy lambda còn tiện lợi hơn rất nhiều khi bạn tìm hiểu hàm **map** (Kteam sẽ giới thiệu ở bài khác).

Câu điều kiện cho lambda

Rõ ràng bạn đã thấy, **lambda** chỉ nhận một expression, do đó, bạn không thể chèn câu lệnh điều kiện như bình thường được mà phải theo một cách khác.

Giả sử với lệnh if như sau

```
if a:
    b
else:
    c
```

Thì có thể viết dưới dạng expression với 2 cách như sau

Cách 1:

```
b if a else c
```

Cách 2:

```
(a and b) or c
```

Bạn có cần nhớ cả 2 cái không? Không cần thiết đâu, Kteam khuyến khích bạn đọc ghi nhớ và dùng cách 1 vì sự rõ ràng và dĩ nhiên cũng không nên bối rối khi thấy cách 2.

Hãy đến với ví dụ để hiểu thêm

```
>>> find_greater = lambda x, y: x if x > y else y
>>> find_greater(1, 3)
3
>>> find_greater(6, 2)
6
```

Ví dụ sau đây là kiểm tra xem số đó cùng có hai ước 2 và 3 hay không? Nếu có thì trả về 1, không thì là 0. Ví dụ này hoàn toàn có thể sử dụng **lambda** bằng cách sử dụng "and" nhưng ở đây Kteam muốn bạn biết chúng ta có thể lồng các expression lên nhau.

```
>>> cd_of_2_3 = lambda x: (1 if x % 3 == 0 else 0) if x % 2 == 0 else 0
>>> cd_of_2_3(6)
1
>>> cd_of_2_3(8)
0
>>> cd_of_2_3(9)
0
>>> cd_of_2_3(12)
1
```

Ở ví dụ trên, phần **if** bạn có thể thu gọn biểu thức đi một tẹo bằng cách dùng phủ định

```
>>> cd_of_2_3 = lambda x: (1 if not (x % 3) else 0) if not (x % 2) else 0
>>> cd_of_2_3(6)
1
>>> cd_of_2_3(9)
0
>>> cd_of_2_3(8)
0
```

```
>>> cd_of_2_3(12)
1
```

Lambda chồng lambda

Phần này sẽ hơi rắc rối nếu như bạn chưa thực sự hiểu **lambda**. Bạn có thể chồng 2 hoặc 3 **lambda** lên nhau cùng một lúc. Nhưng phải chú ý để biết được mình đang làm gì nhé.

```
>>> def kteam(first_string):
...     return lambda second_string: first_string + second_string # trả về một hàm, và lưu biến first_string
...
>>> slogan = kteam('How Kteam ') # gửi giá trị cho biến first_string
>>> slogan
<function kteam.<locals>.<lambda> at 0x00CA4150>
>>> slogan('Free Education') # gửi nốt giá trị còn lại cho second_string
'How Kteam Free Education'
```

Ví dụ trên ta sử dụng "def", và bạn để ý hàm sử dụng "def" trên ta hoàn toàn có thể sử dụng **lambda** thay thế.

```
>>> kteam = lambda first_string: (lambda second_string: first_string + second_string)
>>> slogan = kteam('How Kteam ')
>>> slogan('Free Education')
'How Kteam Free Education'
>>> (lambda first_string: (lambda second_string: first_string + second_string))('How Kteam ')('Free Education')
'How Kteam Free Education'
```

Thực tế thì những **lambda** chồng **lambda** này khá phức tạp. Python vốn không thích sự khó hiểu, phức tạp, sự thiếu thanh lịch thế nên thường thì việc chồng **lambda** như thế này rất không được khuyến khích.

Kết luận

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

Ở bài tiếp theo, Kteam sẽ nói đến MỘT SỐ HÀM HAY SỬ DỤNG KẾT HỢP VỚI 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**".