Bài 15: SỰ KHÁC NHAU VỀ TOÁN TỬ CỦA HASHABLE OBJECT & UNHASHABLE OBJECT TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: Sự khác nhau về toán tử của Hashable object và Unhashable object trong Python

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 các bài trước, Kteam đã giới thiệu đến bạn KIỂU DỮ LIỆU TUPLE, một container thuộc thể loại **hashable object** trong Python

Ở bài này Kteam sẽ nói về **sự khác nhau** của toán tử giữa hai loại kiểu dữ liệu **Hashable Object** (immutable) và **Unhashable Object** (mutable) 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.
- KIỂU DỮ LIỆU SỐ và KIỂU DỮ LIỆU CHUỖI trong Python.
- KIỂU DỮ LIỆU LIST, KIỂU DỮ LIỆU TUPLE trong Python

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

- Giới thiệu cơ bản về hàm id
- Toán tử là một phương thức
- Khác biệt về toán tử Hash Object và Unhash Object
- Tại sao có List lại còn sinh ra Tuple? Hoặc là sử dụng Tuple thôi, cần gì tới List?
- ---

Giới thiệu cơ bản về hàm id

Cú pháp:

```
id(<giá trị>)
```

Như Kteam đã từng đề cập ở các bài trước đây, mọi thứ trong Python xoay quanh các đối tượng, và các giá trị ở đây chính là một đối tượng. Tuy vậy vẫn để là <giá trị> để tránh gây khó hiểu.

Công dụng: Theo định nghĩa về hàm **id** trong tài liệu của Python thì hàm này sẽ trả về một số nguyên (int hoặc longint).

- Giá trị này là một giá trị duy nhất và là hằng số không thay đổi suốt chương trình.
- Trong chi tiết bổ sung của CPython có nói giá trị trả về của hàm id là địa chỉ của giá trị (đối tượng) đó trong bộ nhớ.

Cao siêu là thế, nhưng bạn hoàn toàn có thể nghĩ đơn giản, con số trả về đó như cái số nhà của bạn. Bạn ở đâu, thì số nhà của bạn cũng sẽ tương ứng.

```
>>> n = 69
>>> s = 'How KTeam'
>>> lst = [1, 2]
>>> tup = (3, 4)
>>> id(n)
1446271792
>>> id(s)
53865712
>>> id(lst)
53838352
>>> id(tup)
53865768
>>>
>>> id(123)
1446272656
>>> id('Free Education')
53865832
```

Kteam sẽ tiếp tục giới thiệu hàm id khi nói tới các toán tử so sánh trong Python ở một bài khác.

Toán tử là một phương thức

Lặp lại thêm một lần nữa, mọi thứ xoay quanh Python toàn là hướng đối tượng. Cả các toán tử cũng thế!

```
>>> n = 69
>>> n + 1
70
>>> n
69
>>> n.__add__(1) # tương tự khi bạn n + 1
70
>>> n
69
```

```
>>> n.__sub__(9) # tương tự n - 9
60
>>> n.__mul__(2) # tương tự n * 2
138
>>> n.__radd__(1) # tương tự 1 + n
70
>>> n.__rsub__(9) # tương tự 9 - n
-60
>>> n.__neg__() # tương tự -n
-69
```

Mỗi toán tử của mỗi đối tượng sẽ có toán tử đi kèm.

Khác biệt về toán tử Hash Object và Unhash Object

Vấn đề chính của bài này, là chỉ ra sự khác biệt giữa toán tử ở **hash object** và **unhash object**. Kteam sẽ lấy ví dụ so sánh đơn giản đó chính là sự khác biệt giữa việc **s = s + i** với lại **s += i**

Hãy xem xét đoạn code dưới đây, Kteam sẽ xét một **hash object** là chuỗi:

```
>>> s_1 = 'HowKteam'
>>> s_2 = 'Free Education'
>>> id(s_1)
53866032
>>> id(s_2)
53865712
>>> s_1 = s_1 + ' Python'
>>> s_2 += ' Python'
>>> id(s_1) # đã có sự thay đổi
53866152
>>> id(s_2) # cũng có sự thay đổi
23088304
>>> s_1
'HowKteam Python'
```

```
>>> s_2
'Free Education  Python'
```

Ta cũng thấy, 2 toán tử **= +** cũng không có gì khác biệt lắm so với **+=**.

Giờ ta xét tới một **unhash object**

```
>>> lst_1 = [1, 2]
>>> lst_2 = [3, 4]
>>> id(lst_1)
53839752
>>> id(lst_2)
53864048
>>> lst_1 = lst_1 + [0]
>>> lst_2 += [0]
>>> id(lst_1) # có sự thay đổi
53864088
>>> id(lst_2) # không hề có sự thay đổi
53864048
>>> lst_1
[1, 2, 0]
>>> lst_2
[3, 4, 0]
```

Đã có khác biệt, khi thử với unhash object. Tại sao lại như vậy?

Đó là vì khi bạn làm như cách dưới đây. Tức có nghĩa bạn vừa mới gán lại giá trị cho biến **lst**. Nói cách khác, bạn đã đưa **lst** tới một địa chỉ khác.

```
>>> lst = [1, 2]
>>> lst = lst + [3]
>>> lst
[1, 2, 3]
```

Còn khi bạn làm như thế này

```
>>> lst = [1, 2]
>>> lst += [3]
>>> lst
[1, 2, 3]
```

Thì không như vậy, bạn đã gián tiếp gọi một phương thức

```
>>> lst = [1, 2]
>>> id(lst)
53839752
>>> lst.__iadd__([3])
[1, 2, 3]
>>> id(lst)
53839752
>>> lst
[1, 2, 3]
```

Vậy vì sao, các hash object lại không như vậy?

Là bởi vì các hash object không hề có phương thức **iadd**, hay **imul** như các **unhash object**. Thế nên, khi bạn dùng toán tử **+=**, Python sẽ làm tương tự như bạn dùng cách gán giá trị.

Vì sao các hash object lại không có phương thức iadd, imul?

Khi bạn khởi tạo một giá trị, nó sẽ được lưu trong bộ nhớ máy tính.

- Với **hash object**, bạn không thể thay đổi nội dung của nó. Do đó, Python sẽ xin đủ khoảng trống để lưu trữ dữ liệu của bạn, không nhiều hơn và cũng không ít hơn. Giúp không hoang phí bộ nhớ của bạn. Thế nên, khi bạn cộng thêm một thứ gì đó, Python không biết nhét cái thứ bạn muốn cộng vào chỗ nào. Nên nó đành cuốn gói đi ra chỗ đó, tìm chỗ mới thoáng có đủ khoảng trống.
- Còn với **unhash object**. Là một đối tượng bạn thay đổi được nội dung, vì thế, Python luôn xin dư bộ nhớ để chừa chỗ cho các giá trị tiếp theo bạn có thể thêm vào. Trong bài trước, Kteam đã đề cập đến việc Tuple

chiếm ít dung lượng hơn List vì Tuple là **hash object**. (bạn có thể tham khảo chi tiết tại bài KIỂU DỮ LIỆU TUPLE)

Tại sao có List lại còn sinh ra Tuple? Hoặc là sử dụng Tuple thôi, cần gì tới List?

Đáng lẽ, Kteam sẽ nói vấn đề này ở bài trước, nhưng vì muốn bản hiểu hơn về các **hash object** với **unhash object** nên đã để tới bài này.

Bạn dễ dàng nhận thấy, việc ta thay đổi giá trị của Tuple, không nhất thiết là phải trực tiếp như List.

```
>>> lst = [1, 2]
>>> lst.append(3)
>>> lst
[1, 2, 3]
>>> tup = (1, 2)
>>> tup += (3,)
>>> tup
(1, 2, 3)
```

Các bạn cũng thấy, nó không khác nhau là mấy. Ta cũng có thể tạo ra các hàm thay đổi nội dung của Tuple bằng cách **slicing**. Đã thế List lại còn nặng về việc chiếm nhiều dung lương hơn Tuple, truy xuất chậm hơn Tuple. Việc gì khiến nó còn được trọng dụng?

Vì khi bạn thay đổi Tuple như cách trên, Python phải đi vòng vòng trong bộ nhớ của bạn tìm xem chỗ nào trống, phù hợp để chứa cái Tuple của bạn không, trong khi với List thì không. Do đó, bạn phải biết được dữ liệu của bạn là dạng dữ liệu như thế nào, có cần phải thay đổi không. Dựa vào đó, để chọn ra một kiểu dữ liệu phù hợp cho mình, tối ưu hóa dung lượng sử dụng, thời gian truy xuất.

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài KIỂU DỮ LIỆU TUPLE TRONG PYTHON.

1. Chỉ có **d** là cách khởi tạo đúng. Bạn sẽ hiểu được khái niệm này khi biết tới **Unpacking** và **Packing argument** sẽ được Kteam giới thiệu trong tương lai.
2. c đúng

Nếu bạn thắc mắc vì sao có lỗi. Trong khi ví dụ ở phần "Có phải Tuple luôn là một hash object?" thì lại không có lỗi?

Lí là do vì trong ví dụ phần "Có phải Tuple luôn là một hash object?". Việc thay đổi nội dung List trong Tuple như thế thì Python chỉ làm việc duy nhất với List đó. Không liên quan gì đến Tuple chứa nó.

Riêng ở câu hỏi này, Python đã làm như thế này

- Đưa phần tử tup[2] lên TOS (Top of stack)
- Gán TOS (chính là List [3, 4]) bằng việc cộng thêm cho List đó một List nữa là [50, 60]. Suy ra, List bây giờ đang là [3, 4, 50, 60]
- Sau đó, Python gán lại tup[2] = TOS. Và dĩ nhiên bạn cũng biết, Tuple không thể làm như vậy.

Có thể bạn chưa nắm được kiến thức này, nhưng bạn sẽ thấy nó không hề khó khi đã theo dõi phần hàm id ở đầu bài này.

Kết luận

Bài viết này đã cho bạn biết được cách hoạt động của các toán tử trong Python và một vài sự khác biệt

Ở bài sau, Kteam sẽ nói về một kiểu dữ liệu nữa, đó chính là KIỂU DỮ LIỆU SET 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 quên "**Luyện tập – Thử thách – Không ngại khó**".