**Giới thiệu về phương thức của kiểu dữ liệu List trong Python**

Kiểu dữ liệu List của Python có một số phương thức giúp chúng ta xử lí các vấn đề liên quan đến nó. Kteam sẽ giúp bạn tìm hiểu các phương thức đó.

Một số phương thức Kteam sẽ không nói rõ về nó vì có một số kiến thức bạn chưa nắm được. Điển hình đó là hàm.

Bên cạnh đó có một số phương thức có dạng biến thể là một hàm sẽ được Kteam đề cập ở một bài trong tương lai

**Các phương thức tiện ích**

**Phương thức count**

**Cú pháp:**

<List>**.count(sub, [start, [end]])**

**Công dụng:** Giống với phương thức count của kiểu dữ liệu chuỗi.

* Trả về một số nguyên, chính là số lần xuất hiện của **sub**trong **List**.
* Còn **start**và **end**là số kĩ thuật slicing (lưu ý không hề có bước).

>>> Kteam = [1, 5, 1, 6, 2, 7]

>>> Kteam.count(1)

2

>>> Kteam.count(3)

0

**Phương thức index**

**Cú pháp:**

<List>**.index(sub[, start[, end]])**

**Công dụng:** Tương tự phương thức index của  kiểu dữ liệu chuỗi.

>>> Kteam = [1, 2, 3]

>>> Kteam.index(2)

1

>>> Kteam.index(4)

Traceback (most recent call last):

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

ValueError: 4 **is** **not** **in** list

**Phương thức copy**

**Cú pháp:**

<List>**.copy()**

**Công dụng:**Trả về một List tương tự với List[:]

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

>>> another\_lst = lst.copy() # tương tự lst[:]

>>> another\_lst[0] = 4

>>> another\_lst

[4, 2, 3]

>>> lst

[1, 2, 3]

**Phương thức clear**

**Cú pháp:**

<List>**.clear()**

**Công dụng:** Xóa mọi phần tử có trong List.

**Lưu ý:** Các phiên bản Python 2.X hoặc dưới Python 3.2 sẽ không có phương thức này

>>> Kteam = [1, 2, 3]

>>> Kteam.clear()

>>> Kteam

[]

Phương thức trên bản chất không như những cách gán với một List rỗng. Giống như dưới đây:

>>> lst = []

>>> lst = list()

Phương thức clear sẽ xóa đi các phần tử ở trong List. Các bạn sẽ biết thêm khi biết tới **câu lệnh del**sẽ được Kteam giới thiệu trong các bài sau.

Để thể hiện rõ sự khác biệt giữa hai trường hợp trên. Kteam sẽ lấy ví dụ để minh họa:

* Bạn còn nhớ ví dụ về việc Tèo và gấu của tèo dùng chung số tiền chứ?

>>> tien\_teo = [50]

>>> tien\_teo

[50]

>>> tien\_gau\_cua\_teo = tien\_teo # Tèo và gấu của Tèo đang dùng chung 50 nghìn

>>> tien\_gau\_cua\_teo

[50]

>>> tien\_gau\_cua\_teo = [] # ta gán lại số tiền cô gấu của Tèo là một List rỗng

>>> tien\_gau\_cua\_teo

[]

>>> tien\_teo # và đương nhiên, tiền của Tèo không bị ảnh hưởng

[50]

Tiếp đến, ta sẽ dùng **phương thức clear**.

>>> tien\_teo = [50]

>>> tien\_teo

[50]

>>> tien\_gau\_cua\_teo = tien\_teo # Tèo và gấu của Tèo đang dùng chung 50 nghìn

>>> tien\_gau\_cua\_teo

[50]

>>> tien\_gau\_cua\_teo.clear() # xử dụng phương thức clear

>>> tien\_gau\_cua\_teo

[]

>>> tien\_teo # tiền của Tèo đã bị xóa theo

[]

**Các phương thức cập nhật**

**Phương thức append**

**Cú pháp:**

<List>**.append(x)**

**Công dụng:**Thêm phần tử **x** vào cuối List

>>> howkteam = [1, 2]

>>> howkteam.append(3)

>>> howkteam

[1, 2, 3]

>>> howkteam.append([4, 5]) # chú ý trường hợp này

>>> howkteam

[1, 2, 3, [4, 5]]

**Chú ý:** Đừng bao giờ append một list vào chính nó.

>>> a = [1, 2]

>>> a.append(a)

>>> a

[1, 2, [...]]

Khi ta append một list vào chính nó, thì trên thực tế, nó sẽ tạo ra một vòng lặp vô tận. Trong ví dụ trên, khi ta append a vào list a, thì nó sẽ truy xuất giá trị của a để có thể append. Nhưng vì giá trị của a đang được thay đổi, nên nó sẽ lại append trước khi truy xuất. Điều này sẽ lặp lại mãi mãi vì giá trị a sẽ luôn luôn được thay đổi. Kết quả [1, 2, [...]] chính là đại diện cho sự lặp vô tận đó.

**Phương thức extend**

**Cú pháp:**

<List>**.extend(iterable)**

**Công dụng:** Thêm từng phần tử một của **iterable**vào cuối List.

>>> Kteam = [1, 2, 3]

>>> Kteam.extend([4, 5]) # xem sự khác biệt giữa append và extend

>>> Kteam

[1, 2, 3, 4, 5]

>>> Kteam.extend([[6, 7], 8])

>>> Kteam

[1, 2, 3, 4, 5, [6, 7], 8]

**Phương thức insert**

**Cú pháp:**

<List>**.insert**(**i, x**)

**Công dụng:**Thêm phần **x**vào vị trí **i**ở trong List.

>>> kteam = [1, 2, 3]

>>> kteam.insert(1, 8) # thêm phần tử 8 vào trong List kteam ở vị trí 1

>>> kteam

[1, 8, 2, 3]

Nếu vị trí**i**lại lớn hơn hoặc bằng số phần tử ở trong List thì kết quả sẽ tương tự như **phương thức append**.

>>> kteam= [1, 2, 3]

>>> kteam.insert(4, 20) # vị trí 4, nhưng trong List chỉ có 3 phần tử

>>> kteam

[1, 2, 3, 20]

>>> kteam.insert(len(kteam), 5) # vị trí thứ 4, bằng số phần tử trong List

>>> kteam

[1, 2, 3, 20, 5]

Nếu vị trí **i**là một số âm, bạn cần lưu ý kỹ ví dụ sau. Bạn chắc vẫn còn nhớ về việc indexing với vị trí là một số âm? nếu không nhớ bạn có thể xem lại bài [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 2](http://www.howkteam.vn/course/kieu-du-lieu-chuoi-trong-python--phan-1/kieu-du-lieu-chuoi-trong-python--phan-2-1542) trước khi vào ví dụ này.

>>> kter = [1, 2, 3]

>>> kter[-1]

3

>>> kter[-2]

2

>>> kter[-3]

1

Khi bạn insert mà lại dùng vị trí**i** là số âm, thì vị trí được insert sẽ là **i –1.**

>>> kteam = [1, 2, 3]

>>> kteam[-1]

3

>>> kteam.insert(-1, 4) # thêm vào vị trí (-1 – 1) là -2

>>> kteam

[1, 2, 4, 3]

Nếu vị trí**i –1** (đang xét indexing âm) không có trong List, mặc định, phần tử **x** sẽ được thêm vào đầu List

>>> kteam= [1, 2, 3]

>>> kteam[-20] # không có phần tử -20 trong List

Traceback (most recent call last):

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

IndexError: list index out of range

>>> kteam.insert(-20, 0)

>>> kteam

[0, 1, 2, 3]

**Chú ý:** Cũng giống như phương thức append, Kteam khuyên bạn đừng bao giờ insert một list vào chính nó, bất kể ở vị trí nào.

**Phương thức pop**

**Cú pháp:**

<List>**.pop([i])**

**Công dụng:**Bỏ đi phần tử thứ**i** trong List và trả về giá trị đó. Nếu vị trí **i** không được cung cấp, phương thức này sẽ tự bỏ đi phần tử cuối cùng của List và trả về giá trị đó.

>>> kter= [1, 2, 3, 4, 5, 6]

>>> kter.pop(3)

4

>>> kter

[1,2, 3, 5, 6]

>>> kter.pop(-3)

>>> kter.pop(-3)

3

>>> kter

[1, 2, 5, 6]

>>> kter.pop() # mặc định sẽ pop phần tử cuối cùng nằm trong List

6

>>> kter

[1, 2, 5]

**Phương thức remove**

**Cú pháp:**

<List>**.remove(x)**

**Công dụng:**Bỏ đi phần tử đầu tiên trong List có giá trị x. Nếu trong List không có giá trị x sẽ có lỗi được thông báo

>>> kteam = [1, 5, 6, 2, 1, 7]

>>> kteam.remove(1)

>>> kteam

[5, 6, 2, 1, 7]

>>> kteam.remove(3)

Traceback (most recent call last):

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

ValueError: list.remove(x): x **not** **in** list

**Các phương thức xử lí**

**Phương thức reverse**

**Cú pháp:**

<List>**.reverse()**

**Công dụng:** Đảo ngược các phần tử ở trong List.

>>> kteam= [1, 2, 3]

>>> kteam.reverse()

>>> kteam

[3, 2, 1]

**Phương thức sort**

Phương thức sort giúp ta sắp xếp một list theo một thứ tự nào đó (có thể sắp xếp một cách mặc định, hoặc sắp xếp theo cách mà chúng ta muốn)

**Cú pháp:**

<List>**.sort(key=None, reverse=False)**

**Công dụng:** Sắp xếp các phần tử từ bé đến lớn bằng cách so sánh trực tiếp.

Trong đó:

* Nếu **reverse** là **False**(false là giá trị mặc định), list sẽ được sắp xếp từ bé đến lớn (các phần tử nhỏ hơn sẽ đứng trước), và ngược lại, nếu reverse là **True**, list sẽ được sắp xếp từ lớn đến bé.
* Tham số **key** mặc định là None (tức là sẽ sắp xếp một cách mặc định). Còn nếu key được chỉ định, thì các phần tử được sắp xếp theo cái “quy ước” của key.

**Ví dụ 1:** sắp xếp các list (key = None)

>>> howkteam= [3, 6, 7, 1, 2, 4]

>>> howkeam.sort()

>>> howkteam

[1, 2, 3, 4, 6, 7]

Vì sao nói nó là so sánh trực tiếp. Bởi vì không chỉ số, nó còn so sánh cả chuỗi, cả List, và mọi thứ khác.

>>> lst = ['k', 'free', '9kteam', 'howkteam']

>>> lst.sort()

>>> lst

['9kteam', 'free', 'howkteam', 'k']

Ghi nhớ rằng, các phần tử phải có thể so sánh với nhau. Trường hợp dưới đây bạn không thể so sánh chuỗi với số được, do đó sẽ có lỗi hiện lên.

lst = ['kteam', 69]

>>> lst.sort()

Traceback (most recent call last):

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

TypeError: '<' **not** supported between instances of 'int' **and** 'str'

Chúng ta sẽ nói đến từ khóa **reverse**. Từ khóa này bạn chỉ có thể cho 2 giá trị, một là **True**, hai là **False**.

* Nếu là False, các phần tử được sắp xếp từ bé đến lớn, còn ngược lại là từ lớn đến bé.

>>> kteam = [6, 8, 2, 5, 1, 10, 4]

>>> true\_reverse = kteam.copy() #tạo một bản sao của kteam và không ảnh hưởng đến kteam

>>> kteam.sort() # không đưa giá trị cho reverse thì mặc định là False

>>> true\_reverse.sort(reverse=**True**)

>>> kteam

[1, 2, 4, 5, 6, 8, 10]

>>> true\_reverse

[10, 8, 6, 5, 4, 2, 1]

**Ví dụ 2:** sắp xếp các list (có tham số key được truyền vào)

>>> a = ['This', 'is', 'How', 'Kteam']

>>> b = a.copy()

>>> a.sort()

>>> b.sort(key=len)

>>> b

['is', 'How', 'This', 'Kteam']

>>> a

['How', 'Kteam', 'This', 'is']

Chúng ta có thể thấy rõ ràng rằng: Cùng một giá trị ban đầu, cùng một phương thức sort, nhưng giá trị sau khi sort lại khác nhau hoàn toàn. Nguyên nhân ở đây chính là nằm ở tham số key. Trong ví dụ trên, đối với list a, chương trình sẽ sắp xếp theo mặc định (so sánh trực tiếp 2 chuỗi, đưa chuỗi nhỏ hơn về đầu list). Còn đối với list b, chương trình sẽ sắp xếp dựa trên việc so sánh độ dài của các chuỗi (vì **key = len**, tức là lấy độ dài). Kết quả của b sau khi sort là một list có các chuỗi với độ dài tăng dần.

Bằng cách truyền cho tham số key một phương thức nào đó (hoặc là một hàm – Khái niệm mà Kteam sẽ đề cập trong tương lai), ta có thể để cho một list được sắp xếp theo cách của mình. Ví dụ dưới đây là một cách sắp xếp một list với cả chuỗi và số, dựa trên việc chuyển toàn bộ các số sang kiểu chuỗi và so sánh.

>>> lst = ['a', 1, 'b', 2, 2, 'b']

>>> lst.sort(key=str)

>>> lst

[1, 2, 2, 'a', 'b', 'b']

Vì đây là một phần tương đối mới, nên khi các bạn tiếp cận với khái niệm hàm, Kteam sẽ thêm các ví dụ kèm giải thích để các bạn hiểu rõ hơn.