05-Python数据结构

将其他类型的数据转换为元组：tuple()

In [3]:
```
letters = tuple('hello world')
letters
```

Out[3]: ('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')

In [4]:
```
numbers = tuple(range(5))
numbers
```

Out[4]: (0, 1, 2, 3, 4)

In [5]:
```
keys = tuple({'a': 1, 'b': 2, 'c': 3})
keys
```

Out[5]: ('a', 'b', 'c')

In [6]:
```
values = tuple({'a': 1, 'b': 2, 'c': 3}.values())
values
```

Out[6]: (1, 2, 3)

连接元组：+

In [7]:
```
tup1 = (1, 2, 3)
tup2 = (4, 5, 6)
tup3 = tup1 + tup2
tup3
```

Out[7]: (1, 2, 3, 4, 5, 6)

重复元组：*

In [8]:
```
tup1 * 3
```

Out[8]: (1, 2, 3, 1, 2, 3, 1, 2, 3)

分解元组（unpack）

In [9]:
```
x, y, z = tup1
print(x, y, z)
```

1 2 3

交换变量的值

In [10]:
```
x, y = y, x
print(x, y, z)
```

```
2 1 3
```

统计元组中某元素的出现次数：count(item)

In [11]:
```
tup = (1, 2, 3, 2, 2, 3, 4, 5, 6)
tup.count(2)  # 统计元素2出现的次数
```

Out[11]: 3

应该如何使用元组

- 元组作为数据的记录
- 元组作为不变的列表

In [2]:
```
# Latitude and longitude of Los Angeles International Airport
lax_coordinates = (33.9425, -118.408056)
print(lax_coordinates)
```

```
(33.9425, -118.408056)
```

遍历列表获取元组

In [4]:
```
traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'), ('ESP', 'XDA205856')]
for passport in sorted(traveler_ids):
    print('%s/%s' % passport)
```

```
BRA/CE342567
ESP/XDA205856
USA/31195855
```

元组作为不变的列表：

- 清晰，元组长度不可变
- 性能，元组消耗更少的内存速度更快

In [5]:
```
a = (10, 'alpha', [1, 2])
a[-1].append(99)
a
```

Out[5]: (10, 'alpha', [1, 2, 99])

将元组作为不变的列表不是设计元组的本意(但是这种用法非常普遍)：

> Making tuples behave as sequences was a hack.

列表的连接和重复：+ 和 *

In [1]:
```
lst1 = [1, 2, 3]
lst2 = [4, 5, 6]
lst3 = lst1 + lst2
lst3
```

Out[1]: [1, 2, 3, 4, 5, 6]

In [2]:
```
lst1 * 3
```

Out[2]: [1, 2, 3, 1, 2, 3, 1, 2, 3]

如果重复嵌套的列表会发生什么?

In [4]:
```
nested_lst = [lst1] * 3
nested_lst
```

Out[4]: [[1, 2, 3], [1, 2, 3], [1, 2, 3]]

如果尝试修改嵌套列表中的元素会发生什么?

In [5]:
```
nested_lst[0][0] = 0
nested_lst
```

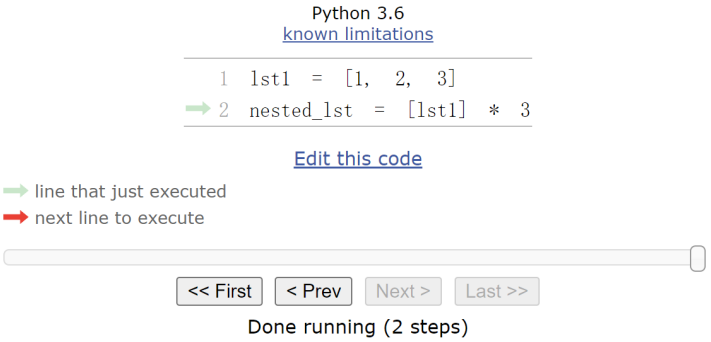
Out[5]: [[0, 2, 3], [0, 2, 3], [0, 2, 3]]

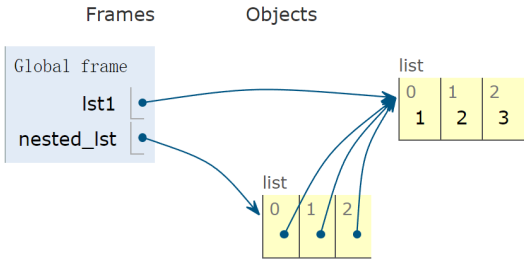
我们本来只想修改nested_lst[0][0]这一个元素，但是每行的第一个元素都被修改了，为什么?

In [7]:
```
print(nested_lst[0] is nested_lst[1]) # 被嵌套的列表的id是相同的
print(nested_lst[0] is nested_lst[2])
```

```
True
True
```

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java





为什么元组相比列表的性能更高?

- 为了评估元组字面值，Python编译器会通过一次操作生成用于元组常量的字节码;
- 而对于列表字面值，生成的字节码会将每个元素作为单独的常量推送到数据栈中，然后再构建列表。
- 给定一个元组t，tuple(t)只会返回对同一t的引用。不需要复制。
- 相比之下，给定一个列表l，list(l)构造函数必须创建l的一个新副本。
- 由于元组具有固定长度，因此分配给元组实例的内存空间正好满足其需求。

- 另一方面，列表实例的分配空间则会多出一些，以摊销未来追加操作的成本。
- 元组中的元素引用保存在元组结构的数组中
- 而列表则保存对另外一个存储在其他地方的引用数组的指针。这种间接引用是必要的，因为当列表增长超出当前分配的空间时，Python需要重新分配引用数组的空间。这种额外的间接引用会减少CPU缓存的效果。

[原文链接](https://stackoverflow.com/questions/68630/are-tuples-more-efficient-than-lists-in-python/22140115#22140115)

In [2]:
```
x = 1 # 1是字面量(literal)
y = 1
print(id(x)) # x和y指向同一个对象
print(id(y))
```

```
2294200402160
2294200402160
```

字典

- 从其他序列数据创建字典
- 读取字典和写字典时的默认值

In [9]:
```
# Creating a dictionary from a list of tuples
pairs = [("a", 1), ("b", 2), ("c", 3)]
dictionary = dict(pairs)
print(dictionary)  # Output: {'a': 1, 'b': 2, 'c': 3}
```

```
{'a': 1, 'b': 2, 'c': 3}
```

In [10]:
```
# Creating a dictionary from two sequences
keys = ["name", "age", "city"]
values = ["John Doe", 25, "New York"]

dictionary = dict(zip(keys, values))
print(dictionary)
```

```
{'name': 'John Doe', 'age': 25, 'city': 'New York'}
```

读取字典时的默认值： `dict.get(key, default=None)`

In [11]:
```
alien_0 = {'color': 'green', 'speed': 'slow'}
# print(alien_0['points'])
print(alien_0.get('points', 'No point value assigned.'))
```

```
No point value assigned.
```

写字典时的默认值： `dict.setdefault()`

In [12]:
```
# 不使用setdefault方法

words = ["apple", "bat", "bar", "atom", "book"]
by_letter = {}
for word in words:
    letter = word[0]
    if letter not in by_letter:
        # 该word的首字母不在字典的keys中
        by_letter[letter] = [word]
    else:
        # 字典已经包含该word首字母的key
        by_letter[letter].append(word)
print(by_letter)
```

```
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

In [17]:
```
by_letter = {}
for word in words:
    letter = word[0]
    print(letter, by_letter.setdefault(letter, []))
    by_letter.setdefault(letter, []).append(word)

print(by_letter)
```

```
a []
b []
b ['bat']
a ['apple']
b ['bat', 'bar']
{'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

集合 set

- 集合是没有顺序（索引）的没有重复元素的数据集。
- 例如字典所有的键可以构成一个集合。
- 集合的元素放在花括号 { } 中表示集合。

In [25]:
```
set([2, 2, 2, 3, 3, 1, 3])
```

Out[25]: {1, 2, 3}

In [31]:
```
{1, 2, 3} == {3, 2, 1}
```

Out[31]: True

空的集合： set()

In [28]:
```
print(set())
print(type({})) # {} is a empty dict
```

```
set()
<class 'dict'>
```

集合的操作和方法

In [29]:
```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}
print(a | b)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

In [30]:
```
print(a & b)
```

```
{3, 4, 5}
```

Table 3-1. Python set operations

Function	Alternative syntax	Description
`a.add(x)`	N/A	Add element `x` to set `a`
`a.clear()`	N/A	Reset set `a` to an empty state, discarding all of its elements
`a.remove(x)`	N/A	Remove element `x` from set `a`
`a.pop()`	N/A	Remove an arbitrary element from set `a`, raising `KeyError` if the set is empty
`a.union(b)`	`a | b`	All of the unique elements in `a` and `b`
`a.update(b)`	`a |= b`	Set the contents of `a` to be the union of the elements in `a` and `b`
`a.intersection(b)`	`a & b`	All of the elements in *both* `a` and `b`
`a.intersection_update(b)`	`a &= b`	Set the contents of `a` to be the intersection of the elements in `a` and `b`
`a.difference(b)`	`a - b`	The elements in `a` that are not in `b`
`a.difference_update(b)`	`a -= b`	Set `a` to the elements in `a` that are not in `b`
`a.symmetric_difference(b)`	`a ^ b`	All of the elements in either `a` or `b` but *not both*
`a.symmetric_difference_update(b)`	`a ^= b`	Set `a` to contain the elements in either `a` or `b` but *not both*
`a.issubset(b)`	`<=`	`True` if the elements of `a` are all contained in `b`
`a.issuperset(b)`	`>=`	`True` if the elements of `b` are all contained in `a`
`a.isdisjoint(b)`	N/A	`True` if `a` and `b` have no elements in common

列表推导

```
[expr for value in collection if condition]
```

In [59]:
```
strings = ["a", "as", "bat", "car", "dove", "python"]
[x.upper() for x in strings if len(x)>2]
```

Out[59]:
```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

嵌入的列表推导

In [61]:
```
all_names = [["John", "Emily", "Michael", "Mary", "Steven"],
             ["Maria", "Juan", "Javier", "Natalia", "Pilar"]]
# 包含两个字母a的姓名
names = [name for names in all_names
               for name in names
               if name.count("a") >= 2]
print(names)
```

```
['Maria', 'Natalia']
```

返回嵌入的列表结构

In [63]:
```
names_lst = [ [name for name in names if name.count("a") >= 2]
               for names in all_names]
print(names_lst)
```

```
[[], ['Maria', 'Natalia']]
```

集合推导

打印名字中包含'SIGN'的字符的集合

In [58]:
```
from unicodedata import name
print({(chr(i),name(chr(i),'')) for i in range(32, 256) if 'SIGN' in name(chr(i),'')})
```

```
{('¬', 'NOT SIGN'), ('÷', 'DIVISION SIGN'), ('¥', 'YEN SIGN'), ('<', 'LESS-THAN SIGN'),
('®', 'REGISTERED SIGN'), ('>', 'GREATER-THAN SIGN'), ('$', 'DOLLAR SIGN'), ('#', 'NUMBER
SIGN'), ('=', 'EQUALS SIGN'), ('¢', 'CENT SIGN'), ('¤', 'CURRENCY SIGN'), ('£', 'POUND SI
GN'), ('+', 'PLUS SIGN'), ('°', 'DEGREE SIGN'), ('µ', 'MICRO SIGN'), ('¶', 'PILCROW SIG
N'), ('×', 'MULTIPLICATION SIGN'), ('±', 'PLUS-MINUS SIGN'), ('©', 'COPYRIGHT SIGN'),
('§', 'SECTION SIGN'), ('%', 'PERCENT SIGN')}
```

字典推导

```
dict_comp = {key-expr: value-expr for value in collection
             if condition}
```

In [65]:
```
# Filtering elements in a dictionary
numbers = {"one": 1, "two": 2, "three": 3, "four": 4, "five": 5}
even_numbers = {key: value for key, value in numbers.items() if value % 2 == 0}
print(even_numbers)
```

```
{'two': 2, 'four': 4}
```

习题：从两个列表来创建字典

有两个列表，可能有不同的长度。第一个由 keys 组成，第二个由 values 组成。

写一个函数 createDict(keys, values)，返回由 keys 和 values 创建的 dictionary。如果没有足够的值，其余的键应该有一个 None 值。如果没有足够的键，就忽略其余的值。

Examples:

```
keys = ['a', 'b', 'c', 'd']
values = [1, 2, 3]
createDict(keys, values) # {'a': 1, 'b': 2, 'c': 3, 'd': None}

keys = ['a', 'b', 'c']
```

In [ ]:
```
def createDict(keys, values):
    return {
        keys[i]: values[i] if i < len(values) else None
        for i in range(len(keys))
    }
```

习题：两个列表的差

你在这个kata中的目标是实现一个差值函数，从一个列表中减去另一个列表并返回结果。 它应该从列表a中删除所有在列表b中存在的值，并保持它们的顺序。

```
array_diff([1,2],[1]) == [2]
```

如果一个值出现在b中，那么它的所有出现必须从另一个中删除。

```
array_diff([1,2,2,2,3],[2]) == [1,3]
```

题目地址 (https://www.codewars.com/kata/523f5d21c841566fde000009)

In [69]:
```
def array_diff(a, b):
    return [x for x in a if x not in b]

print(array_diff([1,2],[1]))
print(array_diff([1,2,2,2,2,3,2,2,4],[2,0]))
```

```
[2]
[1, 3, 4]
```