

# 集合Set

集合，简称集。由任意个元素构成的集体。高级语言都实现了这个非常重要的数据结构类型。

Python中，它是**可变的、无序的、不重复**的元素的集合。

## 初始化

- `set()` -> new empty set object
- `set(iterable)` -> new set object

```
1 s1 = set()
2 s2 = set(range(5))
3 s3 = set([1, 2, 3])
4 s4 = set('abcdabcd')
5
6 s5 = {} # 这是什么?
7 s6 = {1, 2, 3}
8 s7 = {1, (1,)}
9 s8 = {1, (1,), [1]} # ?
```

## 元素性质

- 去重：在集合中，所有元素必须相异
- 无序：因为无序，所以**不可索引**
- 可哈希：Python集合中的元素必须可以hash，即元素都可以使用内建函数hash
  - 目前学过不可hash的类型有：list、set、bytearray
- 可迭代：set中虽然元素不一样，但元素都可以迭代出来

## 增加

- `add(elem)`
  - 增加一个元素到set中
  - 如果元素存在，什么都不做
- `update(*others)`
  - 合并其他元素到set集合中来
  - 参数others必须是可迭代对象
  - 就地修改

```
1 s = set()
2 s.add(1)
3 s.update((1,2,3), [2,3,4])
```

## 删除

- `remove(elem)`
  - 从set中移除一个元素
  - 元素不存在，抛出KeyError异常。为什么是KeyError?
- `discard(elem)`

- 从set中移除一个元素
  - 元素不存在，什么都不做
- pop() -> item
  - 移除并返回任意的元素。为什么是任意元素？
  - 空集返回KeyError异常
- clear()
  - 移除所有元素

```

1 s = set(range(10))
2 s.remove(0)
3 #s.remove(11) # KeyError为什么
4 s.discard(11)
5 s.pop()
6 s.clear()

```

## 修改

集合类型没有修改。因为元素唯一。如果元素能够加入到集合中，说明它和别的元素不一样。

所谓修改，其实就是把当前元素改成一个完全不同的元素，就是删除加入新元素。

## 索引

非线性结构，不可索引。

## 遍历

只要是容器，都可以遍历元素。但是效率都是O(n)

## 成员运算符in

```

1 print(10 in [1, 2, 3])
2 print(10 in {1, 2, 3})

```

上面2句代码，分别在列表和集合中搜索元素。如果列表和集合的元素都有100万个，谁的效率高？

## IPython魔术方法

IPython内置的特殊方法，使用%百分号开头的

- % 开头是line magic
- %% 开头是 cell magic, notebook的cell

```

1 %timeit statement
2 -n 一个循环loop执行语句多少次
3 -r 循环执行多少次loop，取最好的结果
4
5 %%timeit setup_code
6 * code.....

```

```

1 # 下面写一行，列表每次都要创建，这样写不好
2 %timeit (-1 in list(range(100)))
3
4 # 下面写在一个cell中，写在setup中，列表创建一次
5 %%timeit l=list(range(1000000))
6 -1 in l

```

## set和线性结构比较

|  |
|--|
| <pre>%%timeit lst1=list(range(100)) a = -1 in lst1</pre>                                 |
| 1.22 $\mu$ s $\pm$ 4.67 ns per loop (mean $\pm$ std. dev. of 7 runs, 1000000 loops each) |
| <pre>%%timeit lst1=list(range(1000000)) a = -1 in lst1</pre>                             |
| 12 ms $\pm$ 116 $\mu$ s per loop (mean $\pm$ std. dev. of 7 runs, 100 loops each)        |
| <pre>%%timeit set1=set(range(100)) a = -1 in set1</pre>                                  |
| 32 ns $\pm$ 0.141 ns per loop (mean $\pm$ std. dev. of 7 runs, 10000000 loops each)      |
| <pre>%%timeit set1=set(range(1000000)) a = -1 in set1</pre>                              |
| 32.3 ns $\pm$ 0.0916 ns per loop (mean $\pm$ std. dev. of 7 runs, 10000000 loops each)   |

结果说明，集合性能很好。为什么？

- 线性数据结构，搜索元素的时间复杂度是 $O(n)$ ，即随着数据规模增加耗时增大
- set、dict使用hash表实现，内部使用hash值作为key，时间复杂度为 $O(1)$ ，查询时间和数据规模无关，不会随着数据规模增大而搜索性能下降。

## 可哈希

- 数值型int、float、complex
- 布尔型True、False
- 字符串string、bytes
- tuple
- None
- 以上都是不可变类型，称为可哈希类型，hashable

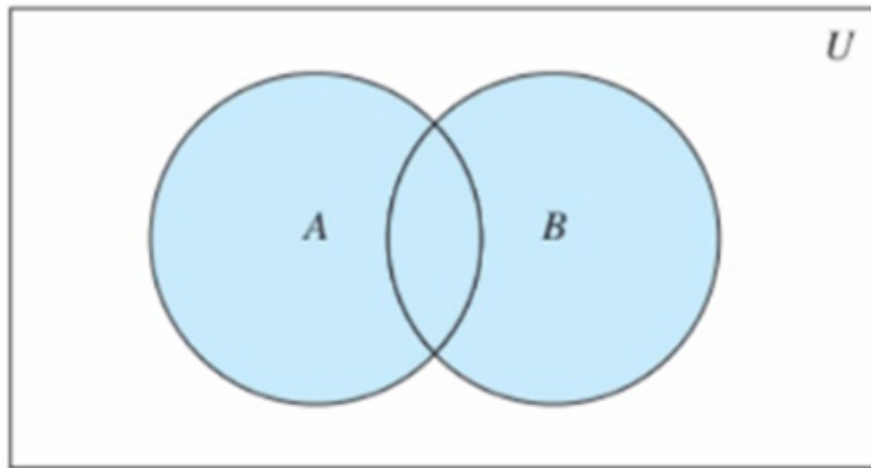
set元素必须是可hash的。

## 集合概念

- 全集
  - 所有元素的集合。例如实数集，所有实数组成的集合就是全集
- 子集subset和超集superset
  - 一个集合A所有元素都在另一个集合B内，A是B的子集，B是A的超集

- 真子集和真超集
  - A是B的子集，且A不等于B，A就是B的真子集，B是A的真超集
- 并集：多个集合合并的结果
- 交集：多个集合的公共部分
- 差集：集合中除去和其他集合公共部分

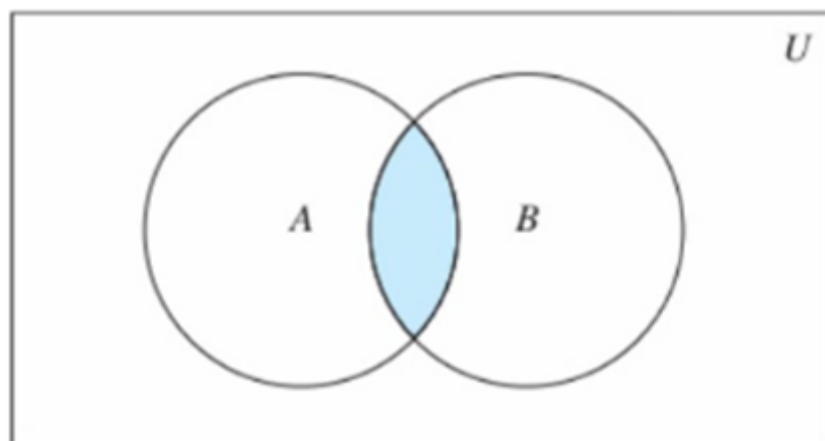
## 并集



将两个集合A和B的所有的元素合并到一起，组成的集合称作集合A与集合B的并集

- `union(*others)` 返回和多个集合合并后的新的集合
- `|` 运算符重载，等同union
- `update(*others)` 和多个集合合并，就地修改
- `|=` 等同update

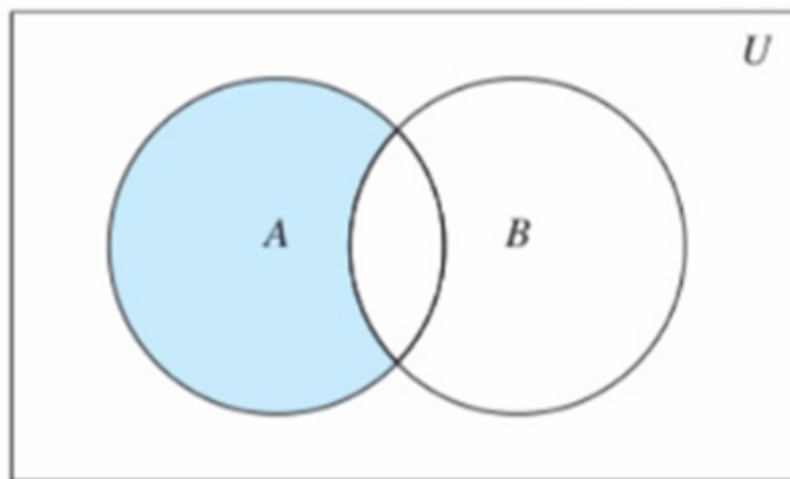
## 交集



集合A和B，由所有属于A且属于B的元素组成的集合

- `intersection(*others)` 返回和多个集合的交集
- `&` 等同intersection
- `intersection_update(*others)` 获取和多个集合的交集，并就地修改
- `&=` 等同intersection\_update

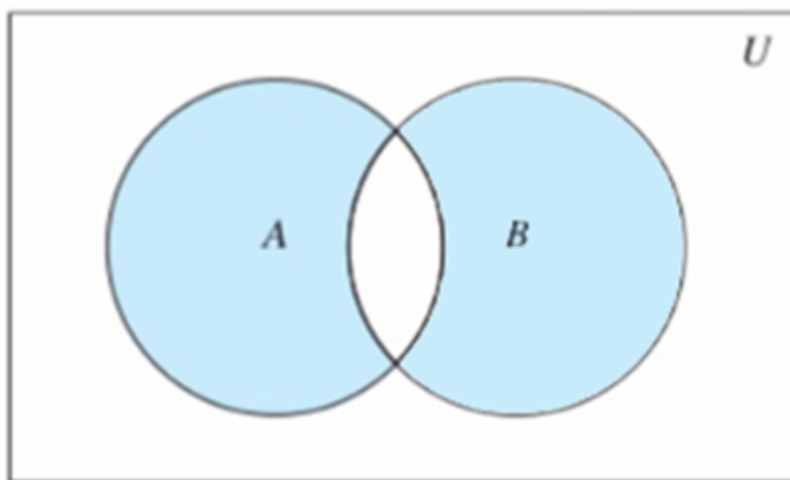
## 差集



集合A和B，由所有属于A且不属于B的元素组成的集合

- `difference(*others)` 返回和多个集合的差集
- `-` 等同difference
- `difference_update(*others)` 获取和多个集合的差集并就地修改
- `-=` 等同difference\_update

## 对称差集



集合A和B，由所有不属于A和B的交集元素组成的集合，记作  $(A-B) \cup (B-A)$

- `symmetric_difference(other)` 返回和另一个集合的对称差集
- `^` 等同symmetric\_difference
- `symmetric_difference_update(other)` 获取和另一个集合的对称差集并就地修改
- `^=` 等同symmetric\_difference\_update

## 其它集合运算

- `issubset(other)`、`<=` 判断当前集合是否是另一个集合的子集
- `set1 < set2` 判断set1是否是set2的真子集
- `issuperset(other)`、`>=` 判断当前集合是否是other的超集
- `set1 > set2` 判断set1是否是set2的真超集
- `isdisjoint(other)` 当前集合和另一个集合没有交集，没有交集，返回True

练习：

- 一个总任务列表，存储所有任务。一个已完成的任務列表。找出为未完成的任務

```
1 业务中，任务ID一般不可以重复
2 所有任务ID放到一个set中，假设为ALL
3 所有已完成的任务ID放到一个set中，假设为COMPLETED，它是ALL的子集
4 ALL - COMPLETED => UNCOMPLETED
```

集合运算，用好了妙用无穷。

## 字典Dict

Dict即Dictionary，也称为mapping。

Python中，字典由任意个元素构成的集合，每一个元素称为Item，也称为Entry。这个Item是由(key, value)组成的二元组。

字典是**可变的、无序的、key不重复**的key-value键值对集合。

### 初始化

- `dict(**kwargs)` 使用name=value对初始化一个字典
- `dict(iterable, **kwarg)` 使用可迭代对象和name=value对构造字典，不过可迭代对象的元素必须是一个二元结构\*\*
- `dict(mapping, **kwarg)` 使用一个字典构建另一个字典

字典的初始化方法都非常常用，都需要会用

```
1 d1 = {}
2 d2 = dict()
3 d3 = dict(a=100, b=200)
4 d4 = dict(d3) # 构造另外一个字典
5 d5 = dict(d4, a=300, c=400)
6 d6 = dict([('a', 100), ('b', 200), (1, 'abc')], b=300, c=400)
```

```
1 # 类方法dict.fromkeys(iterable, value)
2 d = dict.fromkeys(range(5))
3 d = dict.fromkeys(range(5), 0)
```

### 元素访问

- `d[key]`
  - 返回key对应的值value
  - key不存在抛出KeyError异常
- `get(key[, default])`
  - 返回key对应的值value
  - key不存在返回缺省值，如果没有设置缺省值就返回None
- `setdefault(key[, default])`
  - 返回key对应的值value
  - key不存在，添加kv对，value设置为default，并返回default，如果default没有设置，缺省为None

## 新增和修改

- `d[key] = value`
  - 将key对应的值修改为value
  - key不存在添加新的kv对
- `update([other]) -> None`
  - 使用另一个字典的kv对更新本字典
  - key不存在，就添加
  - key存在，覆盖已经存在的key对应的值
  - 就地修改

```
1 d = {}  
2 d['a'] = 1  
3 d.update(red=1)  
4 d.update(['red', 2])  
5 d.update({'red': 3})
```

## 删除

- `pop(key[, default])`
  - key存在，移除它，并返回它的value
  - key不存在，返回给定的default
  - default未设置，key不存在则抛出KeyError异常
- `popitem()`
  - 移除并返回一个任意的键值对
  - 字典为empty，抛出KeyError异常
- `clear()`
  - 清空字典

## 遍历

### 1、遍历Key

```
1 for k in d:  
2     print(k)  
3  
4 for k in d.keys():  
5     print(k)
```

### 2、遍历Value

```
1 for v in d.values():  
2     print(v)  
3  
4 for k in d.keys():  
5     print(d[k])  
6     print(d.get(k))
```

### 3、遍历Item

```

1  for item in d.items():
2      print(item)
3      print(item[0], item[1])
4
5  for k,v in d.items():
6      print(k, v)
7
8  for k,_ in d.items():
9      print(k)
10
11 for _,v in d.items():
12     print(v)

```

Python3中，keys、values、items方法返回一个类似一个生成器的可迭代对象

- Dictionary view对象，可以使用len()、iter()、in操作
- 字典的entry的动态的视图，字典变化，视图将反映出这些变化
- keys返回一个类set对象，也就是可以看做一个set集合。如果values都可以hash，那么items也可以看做是类set对象

Python2中，上面的方法会返回一个新的列表，立即占据新的内存空间。所以Python2建议使用iterkeys、itervalues、iteritems版本，返回一个迭代器，而不是返回一个copy

## 遍历与删除

```

1  # 错误的做法
2  d = dict(a=1, b=2, c=3)
3  for k,v in d.items():
4      print(d.pop(k))

```

在使用keys、values、items方法遍历的时候，不可以改变字典的size

```

1  while len(d):
2      print(d.popitem())
3
4  while d:
5      print(d.popitem())

```

上面的while循环虽然可以移除字典元素，但是很少使用，不如直接clear。

```

1  # for 循环正确删除
2  d = dict(a=1, b=2, c=3)
3  keys = []
4  for k,v in d.items():
5      keys.append(k)
6
7  for k in keys:
8      d.pop(k)

```



# key

字典的key和set的元素要求一致

- set的元素可以就是看做key，set可以看做dict的简化版
- hashable 可哈希才可以作为key，可以使用hash()测试
- 使用key访问，就如同列表使用index访问一样，时间复杂度都是O(1)，这也是最好的访问元素的方式

```
1 d = {  
2     1 : 0,  
3     2.0 : 3,  
4     "abc" : None,  
5     ('hello', 'world', 'python') : "string",  
6     b'abc' : '135'  
7 }
```

## 有序性

字典元素是按照key的hash值无序存储的。

但是，有时候我们却需要一个有序的元素顺序，Python 3.6之前，使用OrderedDict类可以做到，3.6开始dict自身支持。到底Python对一个无序数据结构记录了什么顺序？

```
1 # 3.5如下  
2 C:\Python\Python353>python  
3 Python 3.5.3 (v3.5.3:1880cb95a742, Jan 16 2017, 16:02:32) [MSC v.1900 64 bit  
4 (AMD64)] on win32  
5 Type "help", "copyright", "credits" or "license" for more information.  
6 >>> d = {'a':300, 'b':200, 'c':100, 'd':50}  
7 >>> d  
8 {'c': 100, 'a': 300, 'b': 200, 'd': 50}  
9 >>> d  
10 {'c': 100, 'a': 300, 'b': 200, 'd': 50}  
11 >>> list(d.keys())  
12 ['c', 'a', 'b', 'd']  
13 >>> exit()  
14 C:\Python\Python353>python  
15 Python 3.5.3 (v3.5.3:1880cb95a742, Jan 16 2017, 16:02:32) [MSC v.1900 64 bit  
16 (AMD64)] on win32  
17 Type "help", "copyright", "credits" or "license" for more information.  
18 >>> d = {'a':300, 'b':200, 'c':100, 'd':50}  
19 >>> d  
20 {'b': 200, 'c': 100, 'd': 50, 'a': 300}
```

Python 3.6之前，在不同的机器上，甚至同一个程序分别运行2次，都不能确定不同的key的先后顺序。

```
1 # 3.6+表现如下  
2 C:\Python\python366>python  
3 Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit  
4 (AMD64)] on win32  
5 Type "help", "copyright", "credits" or "license" for more information.  
6 >>> d = {'c': 100, 'a': 300, 'b': 200, 'd': 50}  
7 >>> d  
8 {'c': 100, 'a': 300, 'b': 200, 'd': 50}
```

```
8 >>> exit()
9
10 C:\Python\python366>python
11 Python 3.6.6 (v3.6.6:4cf1f54eb7, Jun 27 2018, 03:37:03) [MSC v.1900 64 bit
    (AMD64)] on win32
12 Type "help", "copyright", "credits" or "license" for more information.
13 >>> d = {'c': 100, 'a': 300, 'b': 200, 'd': 50}
14 >>> d
15 {'c': 100, 'a': 300, 'b': 200, 'd': 50}
16 >>> d.keys()
17 dict_keys(['c', 'a', 'b', 'd'])
```

Python 3.6+, 记录了字典key的**录入顺序**, 遍历的时候, 就是按照这个顺序。

如果使用 `d = {'a':300, 'b':200, 'c':100, 'd':50}`, 就会造成以为字典按照key排序的错觉。

目前, 建议不要3.6提供的这种字典特性, 还是以为字典返回的是无序的, 可以在Python不同版本中考虑使用OrderedDict类来保证这种录入序。