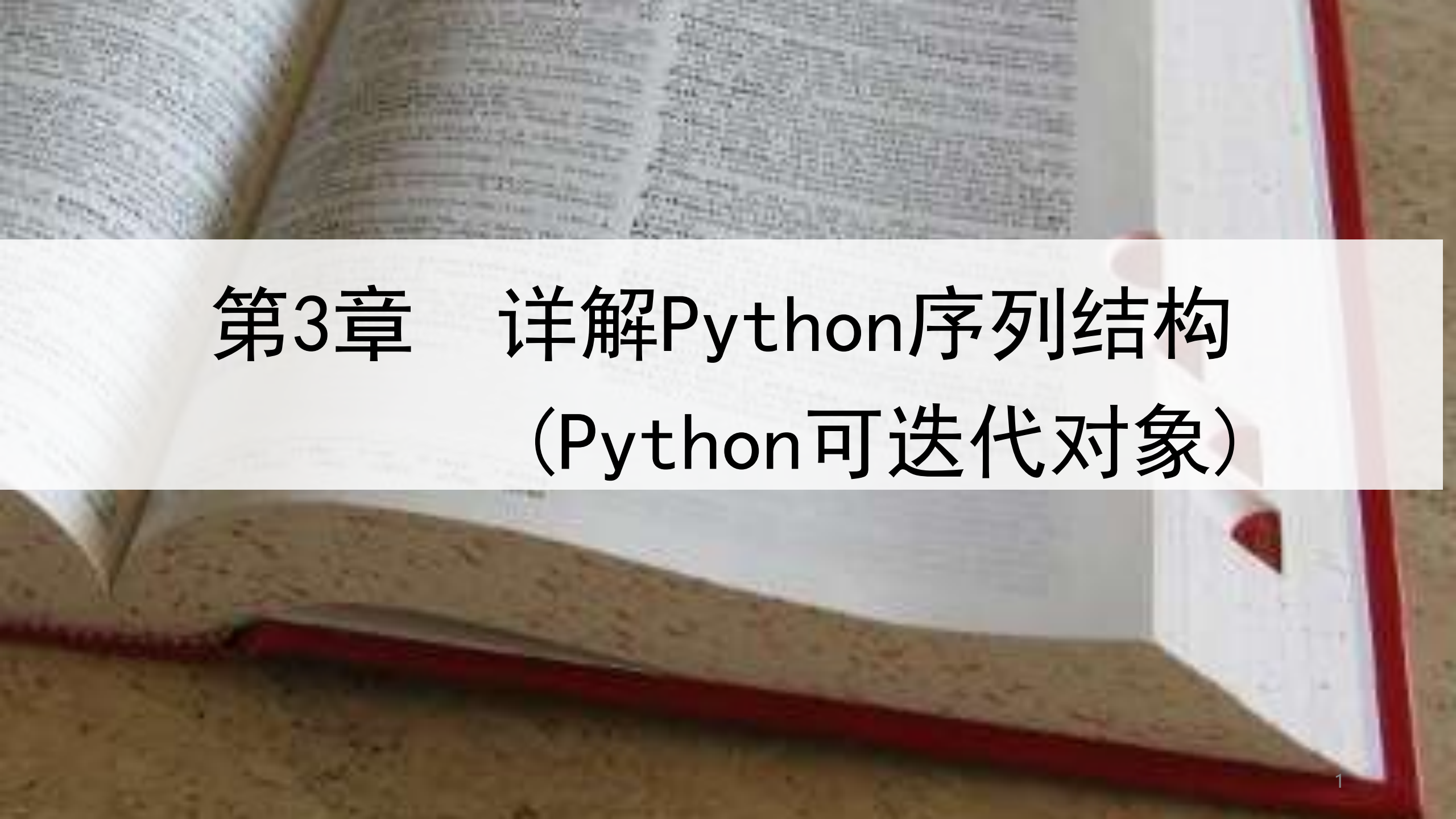
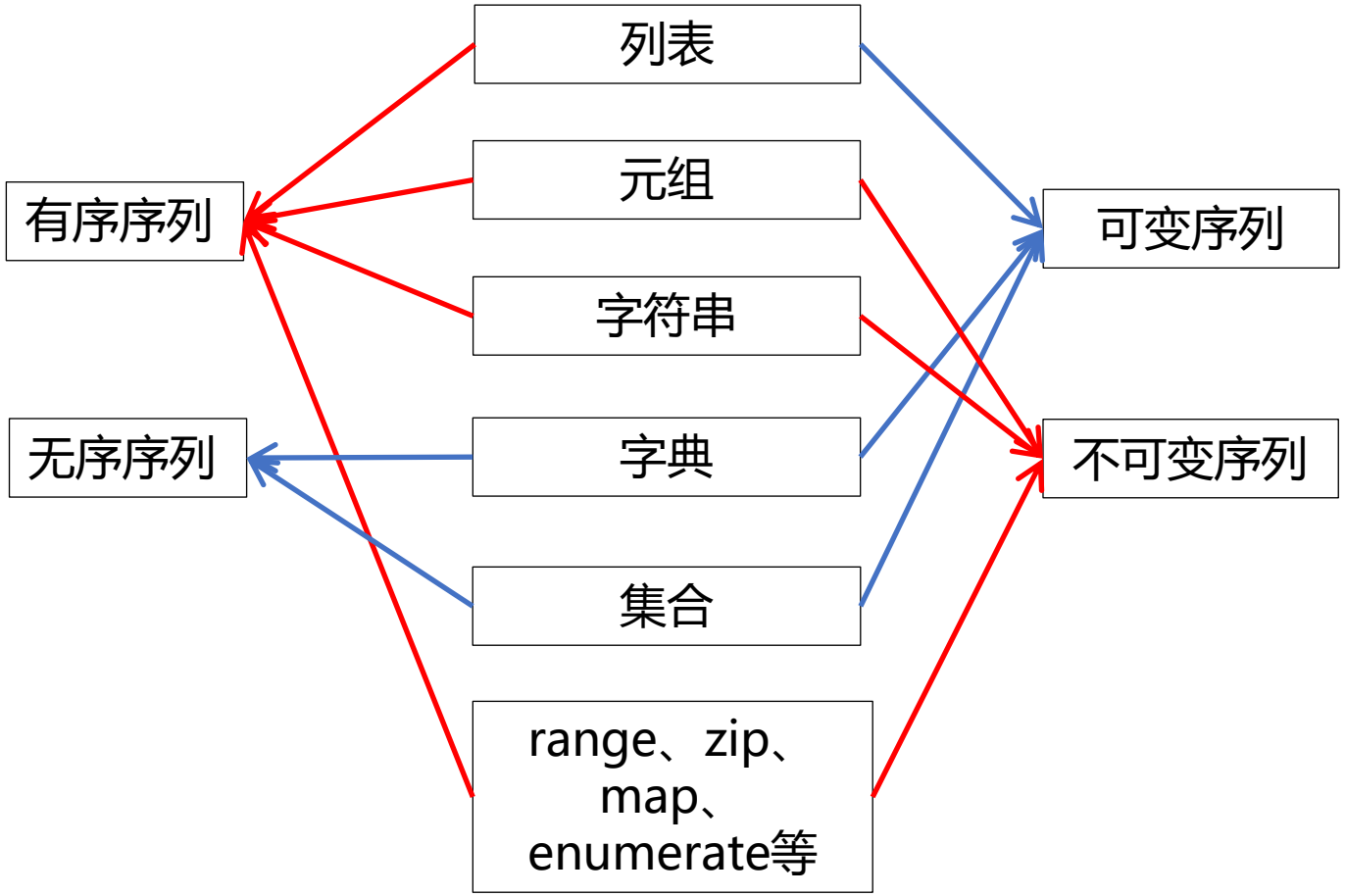


第3章　详解Python序列结构（Python可迭代对象）

第3章 玄之又玄，众妙之门: 详解Python序列结构

- 3.1列表: 打了激素的数组
- 3.2元组: 轻量级列表
- 3.3字典: 反映对应关系的映射类型
- 3.4集合: 元素之间不允许重复

第3章 详解Python序列结构



- 记蓝色的

第3章 详解Python序列结构

	列表(list)	元组(tuple)	字典(dict)	集合(set)
类型名称	list	tuple	dict	set
定界符	方括号[]	圆括号()	大括号{}	大括号{}
是否可变	是	否	是	是
是否有序	是	是	否	否
是否支持下标	是（使用序号作为下标）	是（使用序号作为下标）	是（使用“键”作为下标）	否
元素分隔符	逗号	逗号	逗号	逗号
对元素形式的要求	无	无	键:值	必须可哈希
对元素值的要求	无	无	“键”必须可哈希（不可改变，例：数字、字符串、元组）	必须可哈希
元素是否可重复	是	是	“键”不允许重复，“值”可以重复	否
元素查找速度	非常慢	很慢	非常快	非常快
新增和删除元素速度	尾部操作快 其他位置慢	不允许	快	快

第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - 3.1.2 列表元素访问
  - 3.1.3 列表常用方法
  - 3.1.4 列表对象支持的运算符
  - 3.1.5 内置函数对列表的操作
  - 3.1.6 列表推导式语法与应用案例
  - 3.1.7 切片操作的强大功能

3.1 列表：打了激素的数组

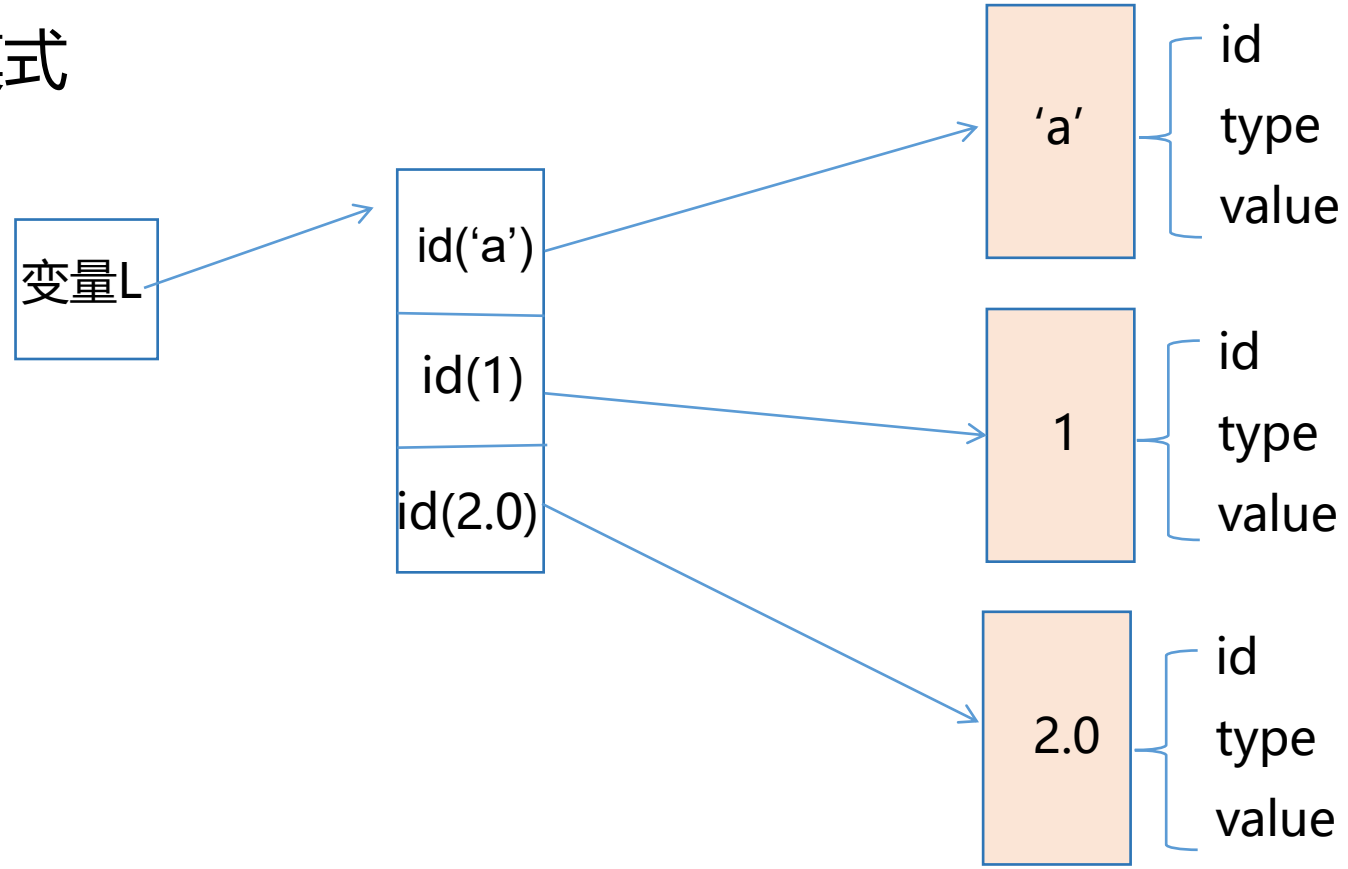
■列表（list）是包含若干元素的**有序连续**内存空间。

■Python采用基于值的自动内存管理模式，变量并不直接存储值，而是存储**值的引用或内存地址**。

■Python列表中的元素也是值的引用，所以列表中各元素可以是不同类型的数据。

3.1 列表：打了激素的数组

列表的内存管理模式



理解：L = ['a',1,2.0]，三个元素（对象）在python中有各自的存储空间，然后列表L只是把L[0]，L[1]，L[2]三个标签贴它们身上，三个标签是连续存储的，最后再用一个变量L指向这三个标签。

3.1 列表：打了激素的数组

- 当列表增加或删除元素时，列表对象自动进行内存的扩展或收缩，从而保证相邻元素之间没有缝隙。

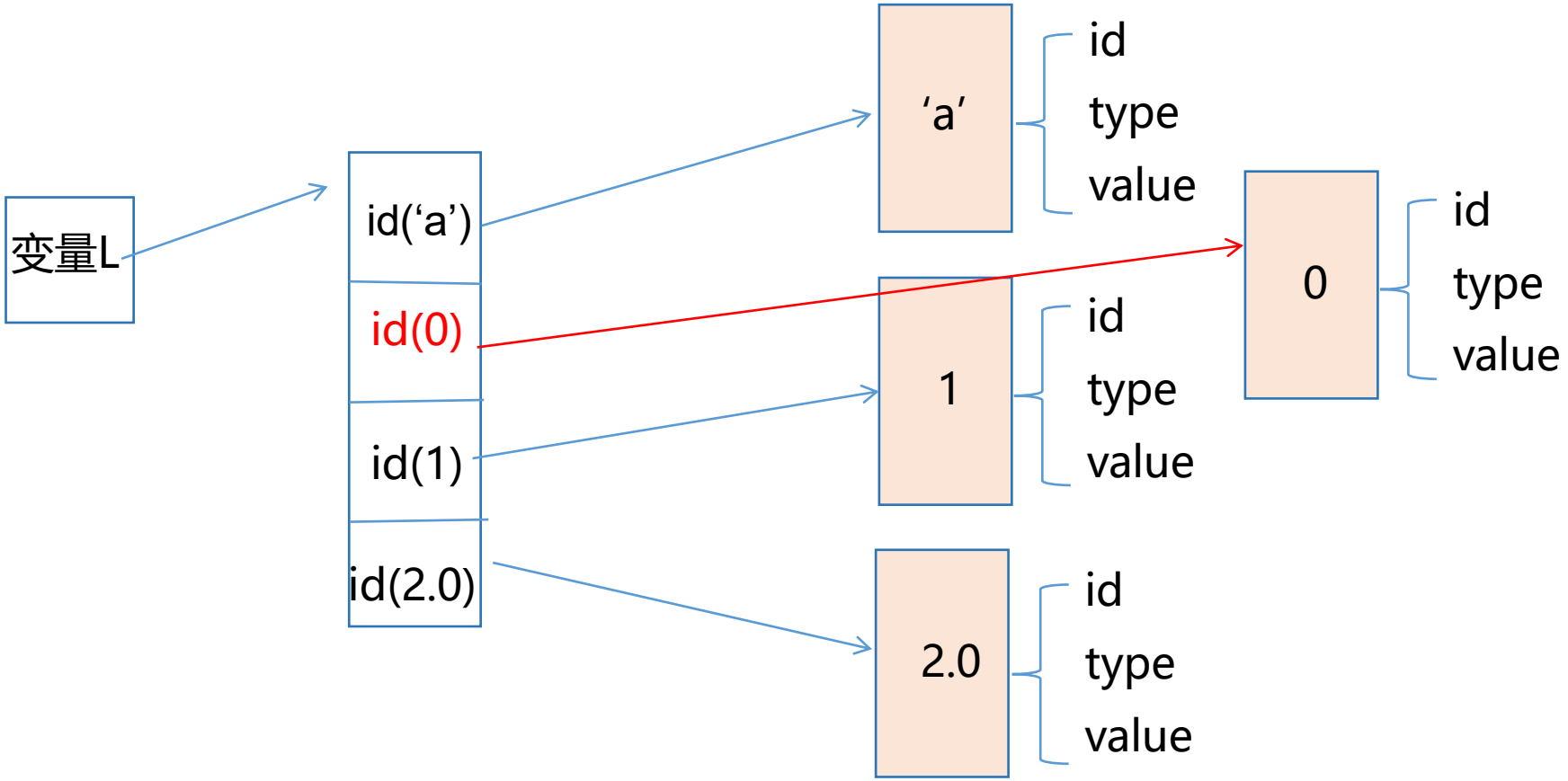
- Python列表的这个内存**自动**管理功能可以大幅度减少程序员的负担，但插入和删除非尾部元素时涉及到列表中大量元素的移动，会严重影响效率。

- 在非尾部位置插入和删除元素时会改变该位置后面的元素在列表中的索引，这对于某些操作可能会导致意外的错误结果。

- 除非确实有必要，否则应尽量从列表尾部进行元素的追加与删除操作。

3.1 列表：打了激素的数组

列表增加中间元素



在L = ['a',1,2.0]中index=1位置插入元素0，则列表自动扩展预备好的很大的空间，先把原来的L[1]，L[2]两个标签复制到新的L[2]和L[3]位置，再把id(0)放在L[1]位置。

3.1 列表：打了激素的数组

- 在Python中，同一个列表中元素的数据类型可以各不相同。
- 列表可以同时包含整数、实数、字符串等基本类型的元素，也可以包含列表、元组、字典、集合、函数以及其他**任意对象**。
- 如果只有一对方括号而没有任何元素则表示**空列表**。
- 列表的功能虽然非常强大，但是开销较大，在实际开发中，最好根据实际的问题选择一种合适的数据类型，要尽量避免过多使用列表？(该用就用)

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
['spam', 2.0, 5, [10, 20]]
[['file1', 200,7], ['file2', 260,9]]
[{3}, {5:6}, (1, 2, 3)]
[]
```

第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - **3.1.1 列表创建与删除**
  - 3.1.2 列表元素访问
  - 3.1.3 列表常用方法
  - 3.1.4 列表对象支持的运算符
  - 3.1.5 内置函数对列表的操作
  - 3.1.6 列表推导式语法与应用案例
  - 3.1.7 切片操作的强大功能

3.1.1 列表创建与删除

- 使用“=”直接将一个列表赋值给变量即可创建列表对象。

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list = []                                  #创建空列表
```

3.1.1 列表创建与删除

- 可以使用list()函数把元组、range对象、字符串、字典、集合或其他可迭代对象转换为列表。

```
>>> list(range(1, 10, 2))                    #将range对象转换为列表
[1, 3, 5, 7, 9]

>>> list('hello world')                      #将字符串转换为列表
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']

>>> list({3,7,5})                            #将集合转换为列表
[3, 5, 7]

>>> list({'a':3, 'b':9, 'c':78})             #将字典的“键”转换为列表
['a', 'c', 'b']

>>> x = list()                                  #创建空列表
```

3.1.1 列表创建与删除

- 当一个列表不再使用时，可以使用del命令将其删除，这一点适用于所有类型的Python对象。

```
>>> x = [1, 2, 3]
>>> y = x                          #复制了标签x
>>> del x                          #删除列表标签
>>> x                              #对象删除后无法再访问，抛出异常
NameError: name 'x' is not defined
>>> y
[1,2,3]                      #del x 只是删除了x标签，不是把[1,2,3]列表删除了

#常用: del x[0]
```

第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - **3.1.2 列表元素访问**
  - 3.1.3 列表常用方法
  - 3.1.4 列表对象支持的运算符
  - 3.1.5 内置函数对列表的操作
  - 3.1.6 列表推导式语法与应用案例
  - 3.1.7 切片操作的强大功能

3.1.2 列表元素访问

- 使用整数作为下标来访问列表的元素，其中0表示第1个元素，1表示第2个元素，2表示第3个元素，以此类推

- 列表还支持使用负整数作为下标，其中-1表示最后1个元素，-2表示倒数第2个元素，-3表示倒数第3个元素，以此类推

- 当下标超过允许的范围时，程序出错：设N为列表中元素的数目，下标的范围是0到N-1和-N到-1，即range(N)和range(-N,0)

序列	元素 1	元素 2	元素 3	元素...	元素 n-1	元素 n
正索引	0	1	2	...	n-2	n-1
负索引	-n	-(n-1)	-(n-2)	...	-2	-1

3.1.2 列表元素访问

```
>>> x = list('Python')                    #创建列表对象
>>> x
['P', 'y', 't', 'h', 'o', 'n']

>>> x[0]                                  #下标为0的元素，第一个元素
'P'
>>> x[-1]                                 #下标为-1的元素，最后一个元素
'n'
```

```
>>> x = list('Python')
>>> x[7]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    x[7]
IndexError: list index out of range
```

Practice

- 设a是一个字符列表，当它的最后一个元素是’n’时，打印a(提示：a可能为空列表)

✓
```
a = list('listen')
if a and a[-1] == 'n':
    print(a)
```

- 哪些内置函数（参考36页表2-5）的返回值为列表？

✓ dir, sorted, list

第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - 3.1.2 列表元素访问
  - **3.1.3 列表常用方法**
  - 3.1.4 列表对象支持的运算符
  - 3.1.5 内置函数对列表的操作
  - 3.1.6 列表推导式语法与应用案例
  - 3.1.7 切片操作的强大功能

3.1.3 列表常用方法

方法	说明
append(x)	将x追加至列表尾部
extend(L)	将列表L中所有元素追加至列表尾部
insert(index, x)	在列表index位置处插入x，该位置后面的所有元素后移并且在列表中的索引加1，如果index为正数且大于列表长度则在列表尾部追加x，如果index为负数且小于列表长度的相反数则在列表头部插入元素x
remove(x)	在列表中删除第一个值为x的元素，该元素之后所有元素前移并且索引减1，如果列表中不存在x则抛出异常
pop([index])	删除并返回列表中下标为index的元素，如果不指定index则默认为-1，弹出最后一个元素；如果弹出中间位置的元素则后面的元素索引减1；如果index不是[-L, L]区间上的整数则抛出异常
clear()	清空列表，删除列表中所有元素，保留列表对象
index(x, [start, [stop]]))	返回列表中第一个值为x的元素的索引，若不存在值为x的元素则抛出异常
count(x)	返回x在列表中的出现次数
reverse()	对列表所有元素进行原地逆序，首尾交换
sort(key=None, reverse=False)	对列表中的元素进行原地排序，key用来指定排序规则，reverse为False表示升序，True表示降序
copy()	返回列表的浅复制（一个新列表）

方法是类内部自带的函数，引用需要声明对象，比如：alist.append()

可输入dir(list)查看list的所有方法

3.1.3 列表常用方法

（1）append()、insert()、extend() —— 增加元素的方法

方法	说明
append(x)	将元素x追加至列表尾部
extend(L)	将列表L中所有元素追加至列表尾部
insert(index, x)	在列表index位置处插入元素x，该位置后面的所有元素后移并且在列表中的索引加1，如果index为正数且大于列表长度则在列表尾部追加x，如果index为负数且小于列表长度的相反数则在列表头部插入元素x

■这3个方法都属于原地操作，不影响列表对象在内存中的起始地址

3. 1. 3　列表常用方法

```
>>> x = [1, 2, 3]
>>> id(x)                          #查看对象的内存地址
50159368

>>> x.append(4)                      #在尾部追加元素
>>> x.insert(0, 0)                  #在指定位置插入元素
>>> x.extend([5, 6, 7])              #在尾部追加多个元素
>>> x
[0, 1, 2, 3, 4, 5, 6, 7]
>>> x.insert(1000,8)

>>> x.append([4])

>>> id(x)                          #列表在内存中的地址不变
50159368

>>> x = [1,2,3]
>>> help(x)                          #查找所有应用
```

3. 1. 3　列表常用方法

append和extend方法的区别：

```
>>> x = [1,2,3]
>>> x.extend('abc')   #extend(L)默认参数是列表，所以先做了list( ‘abc’ )操作
                      #等价于 x.extend(list('abc'))
>>> x
[1, 2, 3, 'a', 'b', 'c']

>>> x = [1,2,3]
>>> x.append('abc')  #append(x)默认x是列表的一个元素
>>> x
[1, 2, 3, 'abc']
```

3.1.3 列表常用方法

（2）pop()、remove()、clear() —— 删除元素的方法

方法	说明
pop([index])	删除**并返回**列表中下标为index的元素，**如果不指定index则默认为-1，弹出最后一个元素**；如果弹出中间位置的元素则后面的元素索引减1；如果index不是[-L, L]区间上的整数则抛出异常
remove(x)	在列表中删除**第一个值为x**的元素，该元素之后所有元素前移并且索引减1，如果列表中不存在x则抛出异常
clear()	清空列表，删除列表中所有元素，保留列表对象

■这3个方法也属于原地操作

■还可以使用del命令删除列表中指定位置的元素，同样也属于原地操作

■使用for value in a遍历列表a时，尽量不remove和insert元素，实际执行过程太过复杂

3.1.3 列表常用方法

```
>>> x = [1, 2, 3, 4, 5, 6, 7]
>>> x.pop()                    #弹出并返回尾部元素
7
>>> x.pop(0)                   #弹出并返回指定位置的元素
1
>>> x.clear()                  #删除所有元素
>>> x
[]


>>> x = [1, 2, 1, 1, 2]
>>> x.remove(2)                #删除首个值为2的元素
>>> del x[3]                   #删除指定位置上的元素
>>> x
[1, 1, 1]

>>> x.remove(3)                #报错
```

3.1.3 列表常用方法

（3）count()、index()—— 元素的访问和计数方法

方法	说明
count(x)	返回x在列表中的出现次数
index(x, [start, [stop]]))	返回列表中第一个值为x的元素的索引，若不存在值为x的元素则抛出异常

```
>>> x = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
>>> x.count(3)                          #元素3在列表x中的出现次数
3
>>> x.count(5)                          #不存在，返回0
0

>>> x.index(2)                          #元素2在列表x中首次出现的索引
1
>>> x.index(5)                          #列表x中没有5，抛出异常
ValueError: 5 is not in list

>>> x.index(value) if value in x else 'None'     #三元运算符
'None'
```

3.1.3 列表常用方法

■index(value, [start, [stop]])，返回下标在start到stop-1范围内，value首次出现的位置

```
>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.index("green")
1
>>> colours.index("green", 2)
3
>>> colours.index("green", 3,4)
3
```

■值在列表中出现的所有位置？

```
colours = ["red", "green", "blue", "green", "yellow"]
indices = [i for i, x in enumerate(colours) if x == 'green']
```

3.1.3 列表常用方法

（4）sort()、reverse()——元素排序的方法

方法	说明
sort(key=None, reverse=False)	对列表中的元素进行原地排序，key用来指定排序规则，reverse为False表示**升序，True表示降序**
reverse()	对列表所有元素进行原地逆序，首尾交换

```
>>> x = list(range(11))                    #包含11个整数的列表
>>> import random
>>> random.shuffle(x)                      #把列表x中的元素随机乱序
>>> x
[6, 0, 1, 7, 4, 3, 2, 8, 5, 10, 9]
```

3.1.3 列表常用方法

```
>>> x.sort()                                    #按默认规则排序，升序
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


>>> x.sort(key=str)                             #按转换为字符串后的大小，升序排序
>>> x
[0, 1, 10, 2, 3, 4, 5, 6, 7, 8, 9]

>>> x.sort(reverse=True)                        #降序排列
>>> x
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

>>> x.reverse()                                 #把所有元素翻转或逆序
>>> x
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

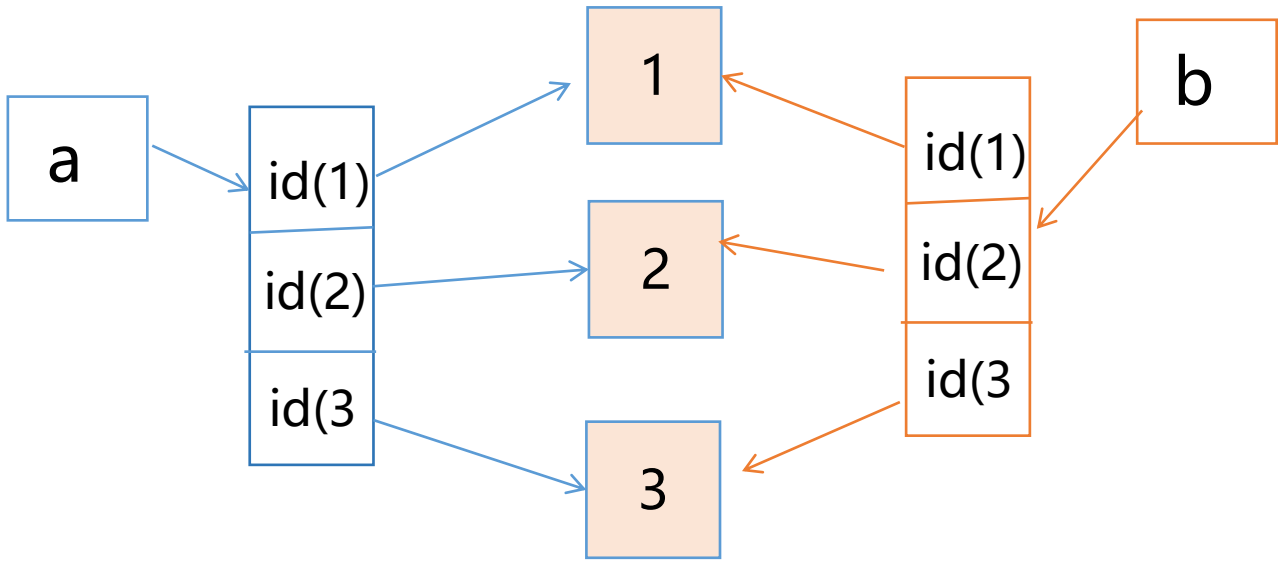
3.1.3 列表常用方法

（5）copy()——列表复制的方法

方法	说明
copy()	返回列表的浅复制（一个新列表）

■当列表中的元素为数字、字符串、元组等不可变类型时，无任何影响

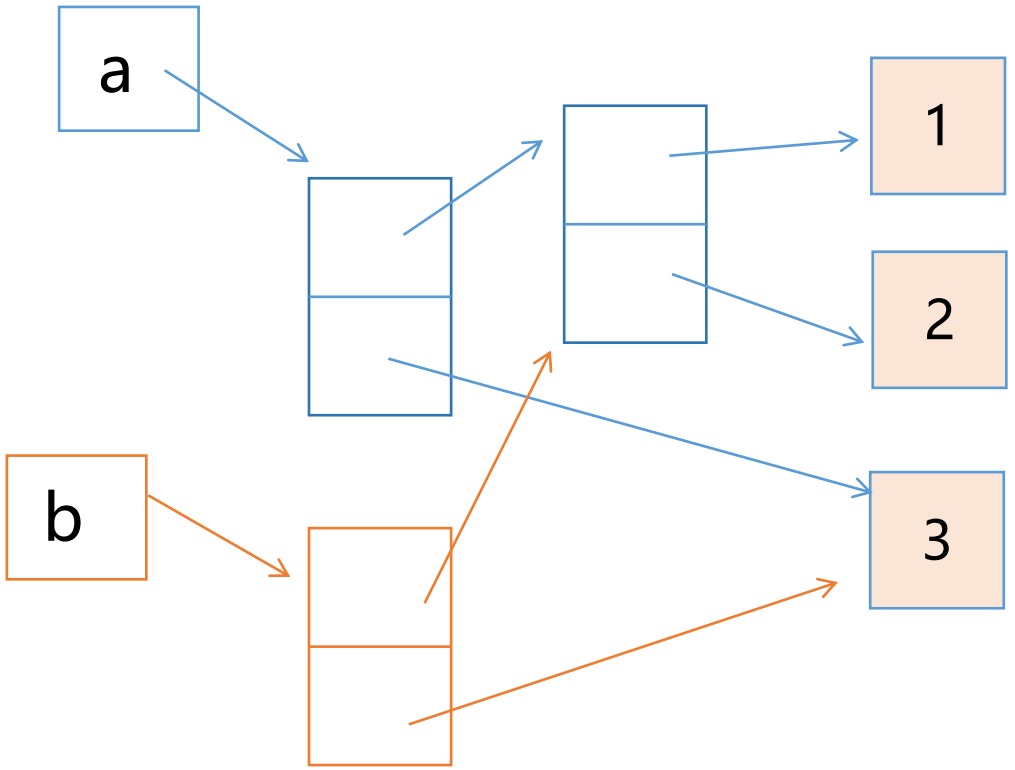
```
>>> a = [1,2,3]
>>> b = a.copy()    #b是新列表
>>> b[0] = 4
>>> b
[4, 2, 3]
>>> a
[1, 2, 3]
```



3. 1. 3　列表常用方法

■ **如果列表元素为列表等可变类型，有影响**

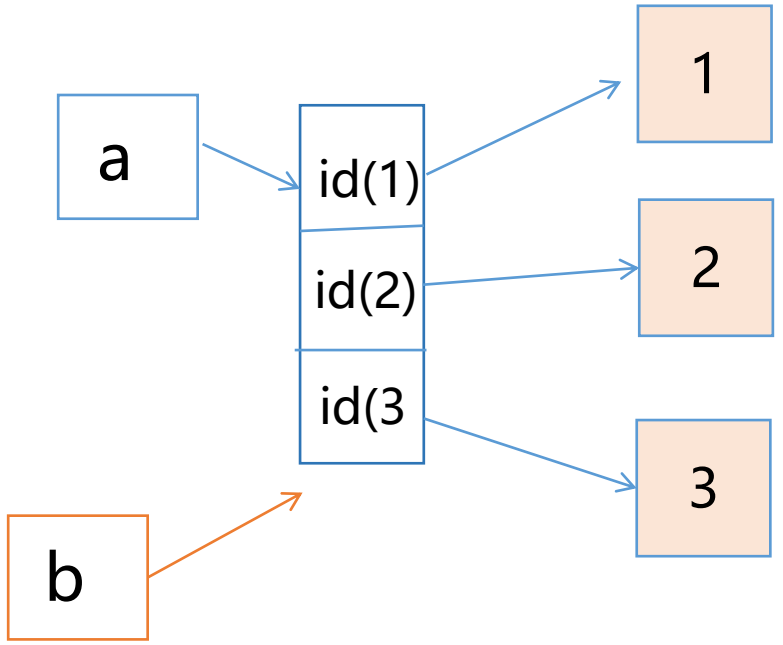
```
>>> a = [[1,2], 3]
>>> b = a.copy()
>>> b[0][0] = 4
>>> b
[[4, 2], 3]
>>> a
[[4, 2], 3]
```



3. 1. 3　列表常用方法

对比：

```
>>> a = [1,2,3]
>>> b = a        #b是新标签
>>> b[0] = 4
>>> b
[4, 2, 3]
>>> a
[4, 2, 3]
```

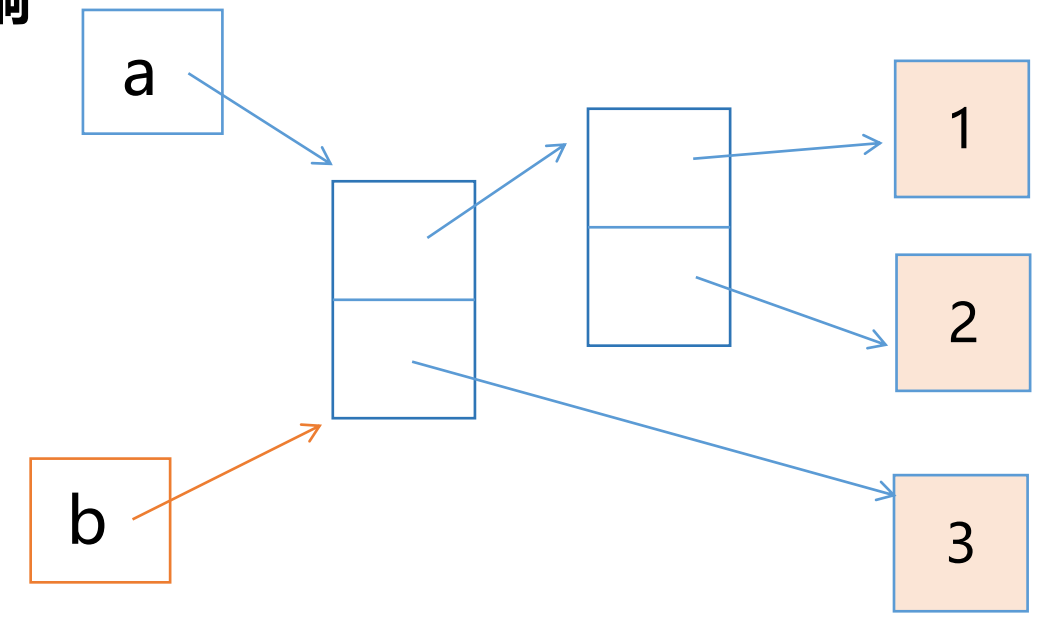


3. 1. 3　列表常用方法

对比：

#如果列表元素为列表等可变类型，有影响

```
>>> a = [[1,2], 3]
>>> b = a
>>> b[0][0] = 4
>>> b
[[4, 2], 3]
>>> a
[[4, 2], 3]
```



第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - 3.1.2 列表元素访问
  - 3.1.3 列表常用方法
  - **3.1.4 列表对象支持的运算符**
  - 3.1.5 内置函数对列表的操作
  - 3.1.6 列表推导式语法与应用案例
  - 3.1.7 切片操作的强大功能

3.1.4 列表对象支持的运算符

- 加法运算符+可以实现列表增加元素的目的，但不属于原地操作，而是返回新列表，涉及大量元素的复制，效率非常低

- 使用复合赋值运算符+=实现列表追加元素时属于原地操作，与append()方法一样高效，等效于extend()方法

3.1.4 列表对象支持的运算符

```
>>> x = [1, 2, 3]
>>> id(x)
53868168

>>> x = x + [4]                    #连接两个列表
>>> x
[1, 2, 3, 4]
>>> id(x)                          #内存地址发生改变
53875720

>>> x += [5]                       #为列表追加元素
>>> x
[1, 2, 3, 4, 5]
>>> id(x)                          #内存地址不变
53875720
```

3.1.4 列表对象支持的运算符

```
import time

n= 100000

start_time = time.time()
a = []
for i in range(n):
    a = a + [i * 2]
end_time = time.time()
print(end_time - start_time)
```

```
start_time = time.time()
a = []
for i in range(n):
    a += [i * 2]
end_time = time.time()
print(end_time - start_time)

start_time = time.time()
a = []
for i in range(n):
    a.append(i * 2)
end_time = time.time()
print(end_time - start_time)
```

```
17.83196735382080
0.018695592880249023
0.011968135833740234
```

3.1.4 列表对象支持的运算符

- 乘法运算符*可以用于列表和整数相乘，表示序列重复（浅复制），返回新列表
- 运算符*=也可以用于列表元素重复，属于原地操作

```
>>> x = [1, 2, 3, 4]
>>> id(x)
54497224
>>> y = x * 2                          #元素重复，返回新列表
>>> y
[1, 2, 3, 4, 1, 2, 3, 4]
>>> id(y)                              #地址发生改变
54603912

>>> y *= 2                             #元素重复，原地进行
>>> y
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> id(x)                              #地址不变
54603912
```

3.1.4 列表对象支持的运算符

- 成员测试运算符in可用于测试列表中是否包含某个元素，查询时间随着列表长度的增加而线性增加
- 同样的操作对于字典和集合而言则是常数级的，与字典集合的大小无关，因为字典和集合用的是散列表

```
>>> 3 in [1, 2, 3]
True
>>> 3 in [1, 2, '3']
False
```

Practice

- 列表的那些操作是在尾部进行的？

✓ append(), extend(), pop(), +=, *=

- sorted()函数与列表的sort()方法的区别是什么？

✓ sorted()返回一个新列表，sort()方法对列表做原地排序

- 在列表a尾部附加一个元素‘ab’的三种方法？

✓ a.append('ab')

✓ a.extend(['ab'])

✓ a += ['ab']

```
>>> a = [1,4,8,2,3]
>>> id(a)
2422438866816

>>> b = sorted(a)
>>> id(b)
2422441956480
```

第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - 3.1.2 列表元素访问
  - 3.1.3 列表常用方法
  - 3.1.4 列表对象支持的运算符
  - **3.1.5 内置函数对列表的操作**
  - 3.1.6 列表推导式语法与应用案例
  - 3.1.7 切片操作的强大功能

3.1.5 内置函数对列表的操作

内置函数的特点：
1.全局可用性，适用于所有序列结构
2.高性能实现(C实现)

■max()、min()函数用于返回列表中所有元素的最大值和最小值

■sum()函数用于返回列表中所有元素之和

■len()函数用于返回列表中元素个数

■all()函数用来测试列表中是否所有元素都等价于True
■any()用来测试列表中是否有等价于True的元素

3.1.5 内置函数对列表的操作

```
>>> x = list(range(11))          #生成列表
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> max(x)
10
>>> min(x)
0
>>> len(x)
11

>>> all(x)                       #测试是否所有元素都等价于True
False
>>> any(x)                       #测试是否存在等价于True的元素
True
```

3.1.5 内置函数对列表的操作

- zip()函数：是迭代器，把多个有序**序列**中**相同位置上的元素**压缩成一个**元组**，所有的元组按顺序形成一个可迭代的zip对象（迭代器），如同拉拉链一样

```
>>> list(zip([1, 2, 3], ['a', 'b', 'c']))  #list把zip对象显示出来
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> for i, value in zip([1, 2, 3], ['a', 'b', 'c']):
        print(i,value)

>>> a1,a2,a3 = 'abc',[1,2,3],'ABC'
>>> for v1,v2,v3 in zip(a1,a2,a2): #可以是多个序列
    print(v1,v2,v3)

>>> a = zip([1, 2], ['a', 'b', 'c'])
>>> list(a)
[(1, 'a'), (2, 'b')]
>>> list(a)
[]    #迭代器只能用一次   注意：range()对象不是迭代器，可以多次使用
```

3.1.5 内置函数对列表的操作

- enumerate()（枚举）函数:是迭代器，用来枚举可迭代对象中的元素，返回可迭代的enumerate对象，其中每个元素都是包含索引和值的**元组**

```
>>> list(enumerate(['a', 'b', 'c']))
[(0, 'a'), (1, 'b'), (2, 'c')]

>>> list(zip(range(3), ['a', 'b', 'c'])) #要多调用一个函数，不如enumerate直观
[(0, 'a'), (1, 'b'), (2, 'c')]
```

```
colours = ["red", "green", "blue", "green", "yellow"]
indices = [i for i, x in enumerate(colours) if x == 'green']
```

3.1.5 内置函数对列表的操作

- map()把一个函数func依次映射到序列的每个元素上，并返回一个可迭代的map对象，map对象中每个元素是原序列中元素经过函数func处理后的结果

```
>>> list(map(str, range(5)))  #把列表中元素转换为字符串
['0', '1', '2', '3', '4']
```

■等价于：

```
a = []
for i in range(5):
    a.append(str(i))
```

```
>>> [ str(i) for i in range(5) ]  #map()是函数式风格，python3.0后大家更喜欢列表推导式
['0', '1', '2', '3', '4']
```

3.1.5 内置函数对列表的操作

```
>>> def add(x, y):              #可以接收2个参数的函数
    return x+y

>>> list(map(add, range(5), range(5,10)))   #把双参数函数映射到两个序列上
[5, 7, 9, 11, 13]            #返回列表

>>> [ i+j for i,j in zip(range(5), range(5,10)) ]
[5, 7, 9, 11, 13]
```

- If Guido van Rossum, the author of the programming language Python, had got his will, this chapter would be missing in our tutorial.
- In his article from May 2005 "All Things Pythonic: The fate of reduce() in Python 3000", he gives his reasons for dropping lambda, map(), filter() and reduce(). He expected resistance from the Lisp and the scheme "folks". What he didn't anticipate was the rigidity of this opposition.
- Enough that Guido van Rossum wrote hardly a year later: "After so many attempts to come up with an alternative for lambda, perhaps we should admit defeat. I've not had the time to follow the most recent rounds, but I propose that we keep lambda, so as to stop wasting everybody's talent and time on an impossible quest."

- We can see the result: lambda, map() and filter() are still part of core Python. Only reduce() had to go; it moved into the module functools.

- His reasoning for dropping them is like this:

✓ There is an equally powerful alternative to lambda, filter, map and reduce, i.e. list comprehension
✓ List comprehension is more evident and easier to understand
✓ Having both list comprehension and "Filter, map, reduce and lambda" is transgressing the Python motto "There should be one obvious way to solve a problem"

如果Python编程语言之父吉多·范罗苏姆（Guido van Rossum）当初能如愿以偿，那么本教程中这一章根本不会存在。
在2005年5月发表的《Pythonic万象：reduce()在Python 3000中的命运》一文中，他阐述了决定舍弃lambda、map()、filter()和reduce()的原因。他预料到会遭到Lisp和Scheme"阵营"的抵制，但没料到反对声浪会如此顽固。以至于不到一年后，吉多就无奈写道："经过多次尝试寻找lambda的替代方案后，也许我们该承认失败了。我最近没时间跟进最新讨论，但我建议保留lambda，别再让大家把才华和时间浪费在这场徒劳的探索上了。"
结果显而易见：lambda、map()和filter()至今仍是Python核心功能，只有reduce()被移到了functools模块。
他主张舍弃这些功能的核心理由在于：

列表解析（list comprehension）能完全替代lambda、filter、map和reduce的功能
列表解析的语法更直观易懂
同时保留列表解析和"filter/map/reduce/lambda"体系，违背了Python"解决问题的方法应该只有一种最直观方式"的设计哲学

（翻译说明：

保留技术术语原貌如"list comprehension"译为行业通用译名"列表解析"
"folks"根据语境译为"阵营"而非字面意义的"伙计们"，体现语言社区的分歧
将英语长句拆分为符合中文阅读习惯的短句结构
重要概念如Python设计哲学用引号强调
补充"核心理由在于"作为逻辑连接词，使技术论点更清晰）

Practice

- 设a是一个列表，使用enumerate()函数实现：打印所有值为v的元素的下标

✓
```
for i,item in enumerate(a):
    if item == v:
        print(i)
```

习题

3.1 表达式 [3] in [1, 2, 3, 4] 的值为_____。

3.2 列表对象的 sort() 方法用来对列表元素进行原地排序，该函数的返回值为_____。

3.3 列表对象的_____方法删除首次出现的指定元素，如果列表中不存在要删除的元素，则抛出异常。

3.14 表达式 [1, 2, 3].count(4) 的值为_____。

3.15 表达式 [1, 2, 3, 1, 2].index(2) 的值为_____。

习题

3.19 多选题：下面的列表方法中有返回值并且返回值不是空值 None 的有（　　）。

A. sort()　　B. pop()　　C. index()　　D. reverse()

3.22 多选题：下面的列表方法中不影响列表内存首地址的有（　　）。

A. sort()　　B. pop()　　C. index()　　D. reverse()

3.23 多选题：下面的列表方法中不影响列表中元素的有（　　）。

A. sort()　　B. count()　　C. index()　　D. pop()

作业4.3.1

1. 使用random库中的choice函数生成包含从’abcd’中选取的10个随机字符的列表a

2. 使用循环与index方法将列表a中’a’的下标存到列表b内

3. 使用循环和remove方法删除列表a中所有的’a’（提示：使用count方法）

第3章 详解Python序列结构

- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - 3.1.2 列表元素访问
  - 3.1.3 列表常用方法
  - 3.1.4 列表对象支持的运算符
  - 3.1.5 内置函数对列表的操作
  - **3.1.6 列表推导式语法与应用案例**
  - 3.1.7 切片操作的强大功能

3.1.6 列表推导式语法与应用案例

■列表推导式使用非常简洁的方式来快速生成满足特定需求的列表

■代码具有非常强的可读性 #体现了python的编程风格

■列表推导式语法形式为：

```
[expression for expr1 in sequence1 if condition1
            for expr2 in sequence2 if condition2
            for expr3 in sequence3 if condition3
            ...
            for exprN in sequenceN if conditionN]
```

■ 最左边的是最外层循环，最右边的为最内层循环

```
a = [expression for v1 in seq1 if cond1
           for v2 in seq2 if cond2]

a = []
for v1 in seq1:
    if cond1:
        for v2 in seq2:
            if cond2:
                a.append(expression)
```

3.1.6 列表推导式语法与应用案例

■列表推导式在逻辑上等价于循环加选择结构，但形式上更加简洁

```
>>> aList = [x*x for x in range(10)]
```

相当于

```
>>> aList = []
>>> for x in range(10):
    aList.append(x*x)
```

3.1.6 列表推导式语法与应用案例

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> aList = [w.strip() for w in freshfruit]
```

- 等价于下面的代码

```
>>> aList = []
>>> for item in freshfruit:
    aList.append(item.strip())
```

3.1.6 列表推导式语法与应用案例

- 阿凡提与国王比赛下棋，国王说要是自己输了的话阿凡提想要什么他都可以拿得出来。阿凡提说那就要点米吧，棋盘一共64个小格子，在第一个格子里放1粒米，第二个格子里放2粒米，第三个格子里放4粒米，第四个格子里放8粒米，以此类推，后面每个格子里的米都是前一个格子里的2倍，一直把64个格子都放满。需要多少粒米呢？

```
>>> sum([2**i for i in range(64)])
18446744073709551615  #约4.6e11吨
>>> int('1'*64, 2)  #生成由64个字符’1’组成的字符串，2代表2进制，转为10进制
18446744073709551615
```

3.1.6 列表推导式语法与应用案例

（1）实现嵌套列表的平铺

```
>>> alist = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [num for sublist in alist for num in sublist]
[1, 2, 3, 4, 5, 6, 7, 8, 9]  #最左边的是最外层循环，最右边的为最内层循环
```

```
>>> alist = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> result = []
>>> for sublist in alist:
        for num in sublist:
            result.append(num)
>>> result
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.1.6 列表推导式语法与应用案例

演示：

```
alist = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
result = []
for sublist in alist:
    for num in sublist:
        result.append(num)
        print(result)
print('最终结果：',result)
```

演示：#子列表深度不同，

```
>>> alist = [[1, 2], [4, 5, 6]]
>>> [num for sublist in alist for num in sublist]
[1, 2, 4, 5, 6]
>>> blist = [[1, 2, 3], 4]     # blist = [[1, 2, 3], [4]]这样可以平铺
#4不是列表，平铺会涉及到广度(4)优先还是深度(1)优先问题，问题变
#复杂，无法用列表推导式
#列表推导式功能有限！
```

3.1.6 列表推导式语法与应用案例

（2）过滤不符合条件的元素

■在列表推导式中可以使用if子句对列表中的元素进行筛选，只保留符合条件的元素

```
>>> alist = [-1, -4, 6, 7.5, -2.3, 9, -11]

>>> [i for i in alist if i>0]                    #所有大于0的数字
[6, 7.5, 9]
```

3.1.6 列表推导式语法与应用案例

（3）同时遍历多个层次的列表或可迭代对象 #要求掌握到**两重**循环

```
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]  # 或 x is not y
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
>>> result = []
>>> for x in [1, 2, 3]:
        for y in [3, 1, 4]:
            if x != y:
            result.append((x,y))
>>> result
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

3.1.6 列表推导式语法与应用案例

演示:

```
a = []
for x in [1, 2, 3]:
    for y in [3, 1, 4]:
        if y!=x:
            a.append((x, y))
        #print(x, y, '**********', a)
print(a)

print([(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y])
```

3.1.6 列表推导式语法与应用案例

（4）列表推导式中可以使用函数或复杂表达式

```
>>> def f(x):
    if x%2 == 0:
        x = x**2
    else:
        x = x+1
    return x

>>> [f(x) for x in [2, 3, 4, -1] if x>0]
[4, 4, 16]

#三元运算符可以作为expression
>>> [x**2 if x%2 == 0 else x+1 for x in [2, 3, 4, -1] if x>0]
[4, 4, 16]
```

3.1.6 列表推导式语法与应用案例

■ **判断**列表中是否所有元素都大于5

```
>>> x = list(range(10))
>>> [item>5 for item in x]  #条件表达式
[False, False, False, False, False, False, True, True, True, True]
```

演示：

```
>>> all([item>5 for item in range(10)])

>>> all([item>5 for item in range(6,10)]) #列表推导式

>>> all((item>5 for item in range(6,10))) #元组相应的推导式叫生成器推导式

>>> all(item>5 for item in range(6,10))  #元组的()可省略
```

3.1.6 列表推导式语法与应用案例

（5）列表推导式搭配enumerate和zip函数

```
>>> colours = ["red", "green", "blue", "green", "yellow"]
>>> colours.index("green")
```

■值在列表中出现的所有位置?

```
indices = [i for i, x in enumerate(colours) if x == 'green']
```

演示:

```
colours = ["red", "green", "blue", "green", "yellow"]
indices = []
for i, x in enumerate(colours):
    if x == 'green':
        indices.append(i)
    print(i, x, '********', indices)
```

3.1.6 列表推导式语法与应用案例

```
>>> def add(x, y):              #可以接收2个参数的函数
    return x+y

>>> list(map(add, range(5), range(5,10)))   #把双参数函数映射到两个序列上
[5, 7, 9, 11, 13]
>>> [ i+j for i,j in zip(range(5), range(5,10)) ]
[5, 7, 9, 11, 13]
```

演示：

```
a = []
for i,j in zip(range(5), range(5,10)):
    a.append(i+j)
    print(i,j, '***********', a)
```

Practice

- 设a是一个列表，写列表推导式，得到：
- a中元素的平方所构成的列表

✓ `[x**2 for x in a]`

- a中的偶数的平方所构成列表

✓ `[x**2 for x in a if x%2 == 0]`

- a中的偶数的下标所构成列表

✓ `[i for i,x in enumerate(a) if x%2 == 0 ]  #可以把 i,x 写成 (i,x)`

- 设a和b是两个列表，写列表推导式，得到
- a和b相同位置上的元素相除所构成的列表

✓ `[x/y for (x,y) in zip(a,b)]`

- a和b相同位置上的元素相除所构成的列表，且判断被除数不能为零

✓ `[x/y for (x,y) in zip(a,b) if y != 0]`

第3章 详解Python序列结构

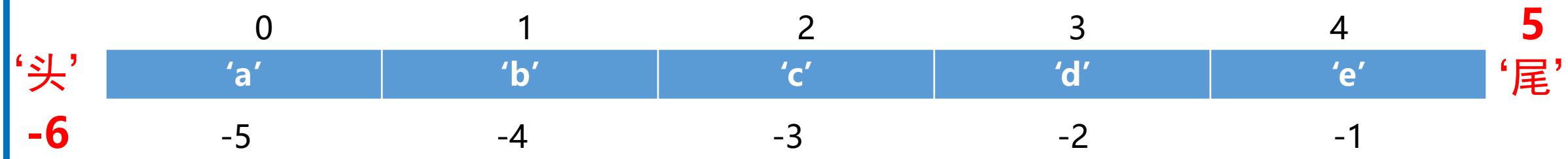
- 3.1 列表：打了激素的数组
  - 3.1.1 列表创建与删除
  - 3.1.2 列表元素访问
  - 3.1.3 列表常用方法
  - 3.1.4 列表对象支持的运算符
  - 3.1.5 内置函数对列表的操作
  - 3.1.6 列表推导式语法与应用案例
  - **3.1.7 切片操作的强大功能**

3.1.7 切片操作的强大功能

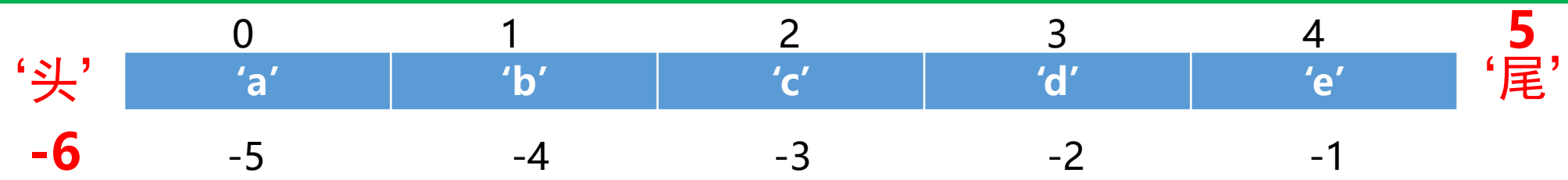
■切片的作用

✓截取序列（有序序列：列表、元组、字符串、range对象）中的一部分得到一个新序列

✓通过切片对列表进行原地操作（修改、增删元素）



3.1.7 切片操作的强大功能



■形式：使用2个冒号分隔的3个数字（只有第一个冒号是必需的）

[start:end:step]

✓第一个数字start表示切片开始位置
- （step>0省略时表示‘头’，step<0省略时表示‘尾’）

✓第二个数字end表示切片截止（但不包含）位置
- （step>0省略时表示‘尾’，step<0省略时表示‘头’）

✓第三个数字step表示切片的步长（默认为1，此时可以省略）
✓省略步长时可以同时省略最后一个冒号
✓start，end可以为负数下标

```
>>>x = ['a', 'b', 'c', 'd', 'e']
>>>x[::2]
['a', 'c', 'e']
>>>x[::-2]
['e', 'c', 'a']

>>>x[0:5:1]
['a', 'b', 'c', 'd', 'e']
>>>x[0:5]
['a', 'b', 'c', 'd', 'e']
>>>x[0:-1]
['a', 'b', 'c', 'd']
```

3.1.7 切片操作的强大功能

```
>>> a = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> len(a)
10

>>> a[1:9:2]
[4, 6, 9, 13]
>>> a[1:-1:2]
[4, 6, 9, 13]
```

```
>>> a[:2]  #与a[0:2]同
[3, 4]
>>> a[-2:] #与a[-2:-1]不同，与a[-2:100]结果相同
[15, 17]

>>> a[:]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> a[::]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
```

3.1.7 切片操作的强大功能

- [start:end:step]当step为负整数时，表示反向切片
- 这时以start为下标的元素应该在以end为下标的元素的右侧（后面）

```
>>> a = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> len(a)
10

>>> a[::-1]                    #start,end全省略，可代表从头到尾取全部
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]

>>> a[9:0:-1]
[17, 15, 13, 11, 9, 7, 6, 5, 4]
>>> a[-1:0:-1]
[17, 15, 13, 11, 9, 7, 6, 5, 4]
>>> a[-1:-11:-1]
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
```

3.1.7 切片操作的强大功能

■ 切片也可用于字符串、元组和range对象，以获取其部分元素

```
>>> '1234'[::-1]
'4321'

>>> (1,2,3,4)[::-1]
(4, 3, 2, 1)

>>> range(1,5)[::-1]
range(4, 0, -1)
```

3.1.7 切片操作的强大功能

（1）使用切片获取列表部分元素

- 使用切片可以返回列表中部分元素组成的新列表
- 切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性

3.1.7 切片操作的强大功能

```
>>> a = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]

>>> a[0:100]              #切片结束位置大于列表长度时，从列表尾部截断
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> a[100]                #抛出异常，不允许越界访问
IndexError: list index out of range

>>> a[100:]               #切片开始位置大于列表长度时，返回空列表
[]

>>> a[100::-1]            #切片逆序，100变成尾部
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]

>>> a[-15:3]              #进行必要的截断处理
[3, 4, 5]
```

Practice

- 设a = [1,2,3,4,5]，下列切片的值?

- a[:-1:2]
- a[-2:]
- a[::-1]
- a[5:]
- a[:0]
- a[3:3]　#step>0时，start>=end,返回[]
- a[3:6]

```
my_list = [1,2,3]
my_list.reverse()  #原地逆序
print(my_list)      #[3,2,1]
```

```
my_list = [1,2,3]
new_list = reversed(my_list)
print(my_list)      #[1,2,3]
print(new_list)     #[3,2,1]
```

- a[::-1]与reversed(a)和a.reverse()的区别?

✓ a[::-1]为逆序组成的新列表;

✓ reversed(a)为逆序组成的reversed对象，不修改原列表，返回一个新的迭代对象;
a.reverse()对a做原地逆序,修改原列表

✓ 对a的逆序做遍历时，应该使用reversed(a)　#因为a.reverse()没有返回值

3.1.7 切片操作的强大功能

（2）使用切片为列表增加元素

■可以使用切片在列表任意位置插入新元素，属于原地操作

■赋值运算符=右侧必须为一个列表或一个序列

```
>>> a = [3, 5, 7]
>>> a[len(a):]          #理解：end=len(a)之后为空
[]
>>> a[len(a):] = [9]       #在列表尾部增加元素 #可用extend()替代
[3, 5, 7, 9]

>>> a[:0]     #理解：start=0之前为空
[]
>>> a[:0] = [1, 2]            #在列表头部插入多个元素的唯一方法
                    #等号右侧也可以为range(1,3)
```

3.1.7 切片操作的强大功能

```
>>> a
[1, 2, 3, 5, 7, 9]

>>> a[3:3]    #理解：index 3 和 3 之间也是空
[]

>>> a[3:3] = [4,4]              #在列表中间位置插入多个元素
>>> a
[1, 2, 3, 4, 4, 5, 7, 9]
```

✓ insert()一次性只能插入一个元素，用上面方法可以一次性插入**多个**元素，这是**一次插入多个元素的唯一方法！**

3.1.7 切片操作的强大功能

（3）使用切片替换列表中的元素

```
>>> a = [3, 5, 7, 9]
>>> a[:3] = [1, 2, 3]          #替换列表元素，等号两边的列表长度相等
>>> a
[1, 2, 3, 9]

>>> a[3:] = [4, 5, 6]          #切片连续，等号两边的列表长度可以不相等
>>> a
[1, 2, 3, 4, 5, 6]

>>> a[::2] = [0]*3             #隔一个修改一个 #长度必须相同，不相等则报错
>>> a
[0, 2, 0, 4, 0, 6]
```

3.1.7 切片操作的强大功能

```
>>> a[::2] = ['a', 'b', 'c']    #隔一个修改一个
>>> a
['a', 2, 'b', 4, 'c', 6]


>>> a[1::2] = range(3)          #序列解包
>>> a
['a', 0, 'b', 1, 'c', 2]

>>> a[::2] = [1]                #切片不连续时等号两边列表长度必须相等
ValueError: attempt to assign sequence of size 1 to extended slice of size 3
```

3.1.7 切片操作的强大功能

（4）使用切片删除列表中的元素

```
>>> a = [3, 5, 7, 9]
>>> a[:3] = []                    #删除列表中前3个元素
>>> a                             #remove(x)只能一次删掉一个元素
[9]
```

■也可以结合使用del命令与切片来删除列表中的部分元素，并且切片元素可以不连续

```
>>> a = [3, 5, 7, 9, 11]
>>> a[::2] = []
ValueError: attempt to assign sequence of size 0 to extended slice of size 3
>>> del a[::2]                    #切片元素不连续，隔一个删一个,只能用del + slice
>>> a
[5, 9]
```

✓切片可以实现批量操作，是很多方法无法做到的！

✓但切片的可读性不如方法。**优先使用方法！**

3.1.7 切片操作的强大功能

操作类型	语法	功能	返回值
基本切片	lst[i:j]	获取索引i到j-1的子列表	新列表
从头切片	lst[:j]	获取开始到j-1的子列表	新列表
到尾切片	lst[i:]	获取索引i到结尾的子列表	新列表
完整切片	lst[:]	获取列表完整副本	新列表
步长切片	lst[::k]	每隔k-1个元素取值	新列表
反转切片	lst[::-1]	获取反转副本	新列表
负索引切片	lst[-i:-j]	使用负索引切片	新列表
切片赋值	lst[i:j] = iterable	替换切片范围	None
切片插入	lst[i:i] = iterable	在位置i插入	None
切片删除	lst[i:j] = []	删除切片范围	None

3. 1　列表总结

1.创建列表的方式：

方法	语法示例	适用场景
字面量	[1, 2, 3]	已知元素的静态列表
list()	list(range(5))	从其他可迭代对象转换
列表推导式	[x**2 for x in range(10)]	需要处理或过滤元素的动态创建
乘法	[0] * 5	创建重复元素的列表
拼接	[1,2] + [3,4] 或 a.extend([3,4])	合并现有列表
copy()	original.copy()	创建列表的浅拷贝
map/filter	list(map(int, ['1', '2', '3']))	函数式编程风格

3. 1　列表总结

2.列表添加元素的方式：

方法	语法	特点	适用场景
append()	lst.append(x)	添加单个元素，原地修改	逐个添加元素
extend()	lst.extend(iterable)	添加多个元素，原地修改	合并列表或可迭代对象
insert()	lst.insert(i, x)	在指定位置插入，原地修改	在特定位置插入一个元素
+	lst1 + lst2	创建新列表，不修改原列表	需要保留原列表的拼接
+=	lst1 += lst2	原地修改，等价于 extend	快速合并，不关心原列表
切片赋值	lst1[i:i] = iterable	灵活插入，原地修改	在中间位置插入多个元素
	lst1[len(lst1):] = iterable	尾部插入，原地修改	在尾部插入多个元素
	lst1[:0] = iterable	头部插入，原地修改	在头部插入多个元素
列表推导式	[x for x in ...]	条件添加，创建新列表	基于条件过滤或转换

3. 1　列表总结

3.列表删除元素的方式：

方法	语法	特点	适用场景
remove()	lst.remove(x)	按值删除第一个匹配项	知道要删除的值
pop()	lst.pop([i])	按索引删除并**返回**元素，无索引时默认删除并返回最后一个	需要获取被删除的元素
del	del lst[i] 或 del lst[i:j:step]	按索引或切片删除	精确控制删除位置和范围
clear()	lst.clear()	清空整个列表	快速清空所有元素
列表推导式	[x for x in lst if condition]	条件删除，创建新列表	基于复杂条件过滤
filter()	filter(func, lst)	函数式条件过滤	使用函数进行条件判断
切片赋值	lst[i:j] = []	替换式删除	灵活的原地修改

3. 1　列表总结

4.列表替换元素的方式：

方法	语法示例	特点	适用场景
索引赋值	lst[i] = new_value	精确控制，原地修改	替换单个或少量元素
切片替换	lst[i:j] = new_list	批量替换，原地修改	替换连续范围的元素
循环替换	for i in range(len(lst)):	条件替换，可控	基于复杂条件的替换
列表推导式	[new if cond else x for x in lst]	函数式，创建新列表	简单条件替换
map函数	list(map(func, lst))	函数式，创建新列表	统一转换操作

3. 1　列表总结

5.列表元素的访问和计数方式：

方法	语法	特点	适用场景
索引访问	lst[i]	快速直接访问	知道确切位置
切片访问	lst[i:j]	获取子列表	需要连续范围的元素
循环访问	for x in lst	遍历所有元素	需要处理每个元素
解包访问	a, b, c = lst	同时赋值多个变量	知道列表确切结构

方法	语法	特点	适用场景
count()	lst.count(x)	简单快速	统计单个元素出现次数
len()	len(lst)	统计元素总数	获取列表长度
Counter	collections.Counter(lst)	功能强大	需要完整频率分析
手动计数	使用列表推导式或 filter()	灵活可控	复杂条件计数

3. 1　列表总结

6.列表元素的排序方式：

方法	语法	特点	内存使用	适用场景
sort()	lst.sort()	原地排序，修改原列表	低	不需要保留原列表
sorted()	sorted(list)	返回新列表，原列表不变	高	需要保留原列表
reverse()	lst.reverse()	原地反转	低	简单反转顺序
reversed()	reversed(list)	返回反转迭代器	低	需要反转迭代器
切片[::-1]	lst[::-1]	创建反转副本	高	需要反转副本

参数	说明	示例
reverse	是否降序排序	sort(reverse=True)
key	排序键函数	sort(key=len)

3. 1　列表总结

7.列表元素的遍历方式：

遍历方法	语法示例	特点	适用场景
直接遍历	for x in lst	简单快速	只需要元素值
enumerate	for i,x in enumerate(lst)	带索引	需要索引和元素值
range+len	for i in range(len(lst))	索引访问	需要修改原列表
zip	for a,b in zip(lst1,lst2)	并行遍历	同时处理多个列表
reversed	for x in reversed(lst)	反向遍历	需要逆序处理
切片遍历	for x in lst[start:end:step]	部分遍历	只需要部分元素
map/filter	for x in map(f, lst)	函数式编程	需要转换或过滤

3.1 列表总结

方法	是否原地操作	语法	功能	返回值
append()	✅ 原地	lst.append(x)	末尾添加元素	None
extend()	✅ 原地	lst.extend(iterable)	扩展列表	None
insert()	✅ 原地	lst.insert(i, x)	指定位置插入	None
remove()	✅ 原地	lst.remove(x)	删除第一个匹配元素	None
pop()	✅ 原地	lst.pop([i])	删除指定位置元素	被删除元素
clear()	✅ 原地	lst.clear()	清空列表	None
sort()	✅ 原地	lst.sort()	排序列表	None
reverse()	✅ 原地	lst.reverse()	反转列表	None
索引赋值	✅ 原地	lst[i] = x	修改单个元素	-
切片赋值	✅ 原地	lst[i:j:step] = iterable	修改切片范围	-

3.1 列表总结

方法	是否原地操作	语法	功能	返回值
sorted()(内置函数)	❌ 非原地	sorted(lst)	对列表排序	新的排序后列表
reversed()(内置函数)	❌ 非原地	reversed(lst)	反转列表顺序	反转迭代器对象
copy()	❌ 非原地	lst.copy()	创建列表的浅拷贝	新的列表副本
切片访问	❌ 非原地	lst[i:j:step]	获取子列表	新的子列表
+ 操作符	❌ 非原地	lst1 + lst2	列表拼接	新的拼接列表
* 操作符	❌ 非原地	lst * n	列表重复	新的重复列表
列表推导式	❌ 非原地	[x for x in lst]	转换或过滤列表	新的列表
map()	❌ 非原地	map(func, lst)	对每个元素应用函数	映射迭代器
filter()	❌ 非原地	filter(func, lst)	过滤列表元素	过滤迭代器

习题

3.4 假设列表对象 aList 的值为 [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]，那么切片 aList[3: 7] 得到的值是_____。

3.8 假设有一个列表 a，现要求从列表 a 中每 3 个元素取 1 个，并且将取到的元素组成新的列表 b，可以使用语句_____。

3.9 使用列表推导式生成包含 10 个数字 5 的列表，语句可以写为_____。

3.11 已知 vec = [[1, 2], [3, 4]]，则表达式 [col for row in vec for col in row] 的值为_____。

3.12 已知 vec = [[1, 2], [3, 4]]，则表达式 [[row[i] for row in vec] for i in range(len(vec[0]))] 的值为_____。

作业4.3.2

1. 设aList = list(range(2,12)),下列切片的值为？
    1. aList[1:9:2]
    2. aList[1:-1:2]
    3. aList[:3]
    4. aList[-3:]
    5. aList[:]
    6. aList[::]
    7. aList[::-1]
    8. aList[9:0:-1]
    9. aList[-1:0:-1]
    10. aList[0:100]
    11. aList[100:]
    12. aList[-15:3]

2. 设a是一个整数列表，写列表推导式，得到或实现：
    ① a中的奇数的平方所构成列表
    ② a中的奇数的下标所构成列表
    ③ 判断a中数是否都是偶数
    ④ 判断a中的数是否至少有一个是偶数

3. 设a和b是两个列表，写列表推导式，得到
    ① a和b相同位置上的元素不相同时相减所构成的列表
    ② 判断a和b相同位置上的元素之差的绝对值是否都小于1.0e-7

第3章 详解Python序列结构

- 3.2 元组：轻量级列表
  - 3.2.1 元组创建与元素访问
  - 3.2.2 元组与列表的异同点
  - 3.2.3 生成器推导式

3.2 元组：轻量级列表

- 列表的功能很强大，但负担也很重，在很大程度上影响了运行效率
- **当仅是对一个有序序列进行遍历或查找，而不需要对其元素进行任何修改时，建议（应该）使用“轻量级的列表”：元组（tuple）**
- 形式上，元组的所有元素放在一对圆括号中，元素之间使用逗号分隔
  - 如果元组中只有一个元素则必须在最后增加一个逗号

第3章 详解Python序列结构

- 3.2 元组：轻量级列表
  - **3.2.1 元组创建与元素访问**
  - 3.2.2 元组与列表的异同点
  - 3.2.3 生成器推导式

3. 2. 1　元组创建与元素访问

```
>>> x = (1, 2, 3)      #直接把元组赋值给一个变量  #>>> type(x)  #tuple

>>> x[0]               #元组支持使用下标访问特定位置的元素
1
>>> x[-1]              #最后一个元素，元组也支持双向索引
3

>>> x[1] = 4           #元组是不可变的
TypeError: 'tuple' object does not support item assignment

>>> x = (3)            #这和x = 3是一样的  #>>> type(x)  #int

>>> x
3
>>> x = (3,)           #如果元组中只有一个元素，必须在后面多写一个逗号
                       #>>> type(x)  #tuple
>>> x
(3,)
```

3.2.1 元组创建与元素访问

```
>>> x = ()                  #空元组    #>>> type(x)  #tuple

>>> x = tuple()             #空元组

>>> tuple(range(5))         #将其他迭代对象转换为元组
(0, 1, 2, 3, 4)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple([1,2,3])
(1, 2, 3)
>>> tuple(map(str,range(5)))
('0', '1', '2', '3', '4')
```

3.2.1 元组创建与元素访问

```
>>> x = (1, 2, 3)
>>> y = x                #复制标签
>>> y is x
True
>>> del x                #只删除标签
>>> x
NameError: name 'x' is not defined
>>> y
(1, 2, 3)
>>> del x[0]             #不能删除某个元素
NameError: name 'x' is not defined
```

3. 2. 1　元组创建与元素访问

- 很多内置函数的返回值也是包含了若干元组的可迭代对象，例如enumerate()、zip()等等。

```
>>> list(enumerate(range(5)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
>>> list(zip(range(3), 'abcdefg'))
[(0, 'a'), (1, 'b'), (2, 'c')]
```

#enumerate(),zip()它们返回的是一个迭代器（标签），而不是具体的序列，需用list(enumerat()),tuple(enumerate())等将序列(标签对应的值)显示出来

第3章 详解Python序列结构

- 3.2 元组：轻量级列表
  - 3.2.1 元组创建与元素访问
  - **3.2.2 元组与列表的异同点**
  - 3.2.3 生成器推导式

3.2.2 元组与列表的异同点

```
>>> dir(list)
[..., 'append', 'clear', 'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse', 'sort']

>>> dir(tuple)
[..., 'count', 'index']
```

注意：元组没有 copy() 方法：对于不可变对象，一般无需复制操作，若需要直接赋值即可，"简单优于复杂"

```
>>>a = (1,2,3)
>>>b = a  #元组复制
>>>b is a
True
```

```
>>>a = (1,2,3)
>>>b = a[:]  #切片复制
>>>b is a
True
```

```
>>>a = (1,2,3)
>>>b = tuple(a)  #函数复制
>>>b is a
True
```

3. 2. 2　总结元组的所有操作

类别	方法/操作	语法	描述	示例
创建元组	空元组	() 或 tuple()	创建空元组	t = ()
	单元素元组	(item,) 或 item,	创建单元素元组	t = (1,) 或 t = 1,
	多元素元组	(item1, item2)	创建多元素元组	t = (1, 2, 3)或 t=1,2,3
	从序列转换	tuple(iterable)	从其他序列转换	t = tuple([1, 2, 3])
访问元素	索引访问	t[index]	通过索引访问元素	t[0]
	负索引	t[-index]	从末尾开始访问	t[-1]
	切片	t[start:end:step]	获取子元组	t[1:3]

3.2.2 总结元组的所有操作

类别	方法/操作	语法	描述	示例
查询操作	元素存在	item in t	检查元素是否存在	'a' in t
	元素不存在	item not in t	检查元素是否不存在	'x' not in t
	计数	t.count(item)	统计元素出现次数	t.count('a')
	查找索引	t.index(item)	返回元素首次出现的索引	t.index('b')
	指定范围查找	t.index(item, start, end)	在指定范围内查找	t.index('b', 2, 5)
基本信息	长度	len(t)	返回元组长度	len(t)
	最小值	min(t)	返回最小元素	min(t)
	最大值	max(t)	返回最大元素	max(t)
	求和	sum(t)	返回所有元素的和	sum(t)
	枚举遍历	for i, item in enumerate(t):	带索引遍历	for i, x in enumerate(t):

3.2.2 总结元组的所有操作

类别	方法/操作	语法	描述	示例
组合操作	连接	t1 + t2	连接两个元组	(1,2) + (3,4) → (1,2,3,4)
	重复	t * n	重复元组n次	(1,2) * 3 → (1,2,1,2,1,2)
比较操作	相等	t1 == t2	比较是否相等	(1,2) == (1,2) → True
	大小	t1 < t2	按字典序比较	(1,2) < (1,3) → True
解包操作	基本解包	a, b = t	将元组解包到变量	a, b = (1, 2)
	星号解包	a, *b, c = t	扩展解包	a, *b, c = (1,2,3,4)
遍历操作	直接遍历	for item in t:	遍历元组元素	for x in t: print(x)
	枚举遍历	for i, item in enumerate(t):	带索引遍历	for i, x in enumerate(t):
转换操作	转列表	list(t)	转换为列表	list((1,2,3)) → [1,2,3]
	转字符串	str(t)	转换为字符串	str((1,2)) → "(1, 2)"
排序操作	排序	sorted(t)	返回排序后的列表	sorted((3,1,2)) → [1,2,3]
	反向排序	sorted(t, reverse=True)	返回降序排序列表	sorted((3,1,2), reverse=True) → [3,2,1]

3. 2. 2　元组与列表的异同点

- 同：
  - 列表和元组都属于有序序列，都支持使用双向索引访问其中的元素，都支持使用切片截取一部分元素得到一个新序列
  - 都可使用count()方法统计指定元素的出现次数和index()方法获取指定元素的索引
  - 都支持len()、map()、filter()等大量内置函数和+、+=、in等运算符

3.2.2 元组与列表的异同点

■异：

- 元组属于不可变（immutable）序列，不可以直接修改元组中元素的值，也无法为元组增加或删除元素。
- 元组没有提供append()、extend()、insert()、remove()和pop()方法，也不支持对元组元素进行del操作，而**只能使用del命令删除整个元组。**
- 元组也支持切片操作，而不允许使用切片来修改元组

3.2.2 元组与列表的异同点

- Python的内部实现对元组做了大量优化，访问速度比列表更快
- 元组在内部实现上不允许修改其元素值，从而使得代码更加安全，例如调用函数时使用元组传递参数可以防止在函数中修改元组，而使用列表则很难保证这一点

3.2.2 元组与列表的异同点

特性	元组 (Tuple)	列表 (List)	说明与示例
基本定义	不可变序列	可变序列	元组创建后不能修改，列表可以
语法表示	(1, 2, 3) 或 1, 2, 3	[1, 2, 3]	元组括号可省略，列表不可
空序列	() 或 tuple()	[] 或 list()	
单元素	(1,) 或 1,	[1]	元组单元素必须加逗号
可变性	❌ 不可变	✅ 可变	元组创建后不能增删改
哈希性	✅ 可哈希	❌ 不可哈希	元组可作为字典键，列表不可
内存占用	较小	较大	元组更节省内存
性能	创建和访问更快	修改操作更快	元组适合只读场景

3. 2

操作类型	元组	列表	说明
创建	t = (1, 2, 3)	lst = [1, 2, 3]	
访问	t[0] ✅	lst[0] ✅	都支持索引访问
切片	t[1:3] ✅	lst[1:3] ✅	都支持切片
修改元素	t[0] = 1 ❌	lst[0] = 1 ✅	元组不可修改
添加元素	❌ 不支持	append(), insert() ✅	
删除元素	❌ 不支持	remove(), pop() ✅	
连接	+ ✅	+ ✅	都支持连接
重复	* ✅	* ✅	都支持重复
长度	len() ✅	len() ✅	
包含检查	in ✅	in ✅	
计数	count() ✅	count() ✅	
查找索引	index() ✅	index() ✅	
排序	sorted() ✅	sorted() ✅ sort() ✅	元组返回新列表
反转	reversed() ✅	reversed() ✅ reverse() ✅	元组返回迭代器
复制	直接赋值 ✅	copy() ✅ 或直接赋值	元组无需复杂的拷贝

3. 2. 2　元组与列表的异同点

特殊方法	元组	列表	描述
append()	❌	✅	列表末尾添加元素
extend()	❌	✅	扩展列表
insert()	❌	✅	插入元素
remove()	❌	✅	删除指定元素
pop()	❌	✅	删除并返回元素
clear()	❌	✅	清空列表
sort()	❌	✅	原地排序
reverse()	❌	✅	原地反转
copy()	❌	✅	浅拷贝

3.2.2 元组与列表的异同点

使用场景	推荐使用	理由
数据保护	元组 ✅	不可变性保证数据安全
字典键	元组 ✅	可哈希性
函数多返回值	元组 ✅	轻量且安全
常量集合	元组 ✅	语义明确
动态数据集合	列表 ✅	需要频繁修改
栈/队列	列表 ✅	支持增删操作
数据缓存	元组 ✅	内存效率高
迭代操作	元组 ✅	性能更好

第3章 详解Python序列结构

- 3.2 元组：轻量级列表
  - 3.2.1 元组创建与元素访问
  - 3.2.2 元组与列表的异同点
  - **3.2.3 生成器推导式**

3.2.3 生成器推导式

```
g = ((i+2)**2 for i in range(10))
```

- **生成器推导式**（generator expression），在形式上使用圆括号作为定界符，内容与列表推导式相同
- 与列表推导式**最大的不同**是，生成器推导式的结果是一个生成器对象
- 生成器对象是一种可迭代对象，但具有**惰性求值**的特点，只在需要时生成新元素，并不事先存储所有元素
- 比列表推导式具有更高的效率，空间占用非常少，因此**应优先使用**

- *生成器推导式是创建生成器的方式之一。*

3. 2. 3　生成器推导式

生成器是一种特殊的迭代器(遍历器)，它不会一次性生成所有数据，而是按需生成（惰性求值），从而节省内存。

生成器的特点：

- 惰性计算：只在需要时生成值
- 内存高效：不会一次性存储所有数据
- 一次性使用：生成器只能迭代一次
- 状态保持：记住上次执行的位置

性能指标	生成器	迭代器
开发效率	高（代码简洁）	低（需要更多代码）
执行效率	相当	相当
内存效率	高	取决于实现
可读性	高	低

生成器是更强大、更简洁的迭代器，在大多数情况下是首选！

3. 2. 3　生成器推导式

```
>>> a = [0,1,2,3]
>>> ls = [(i+2)**2 for i in a]     #type(ls) ->list
[4, 9, 16, 25]

>>> g = ((i+2)**2 for i in a)     #type(g)->generator
>>> next(g)
4
>>> next(g)
9
```

3.2.3 生成器推导式

生成器推导式 vs 列表推导式

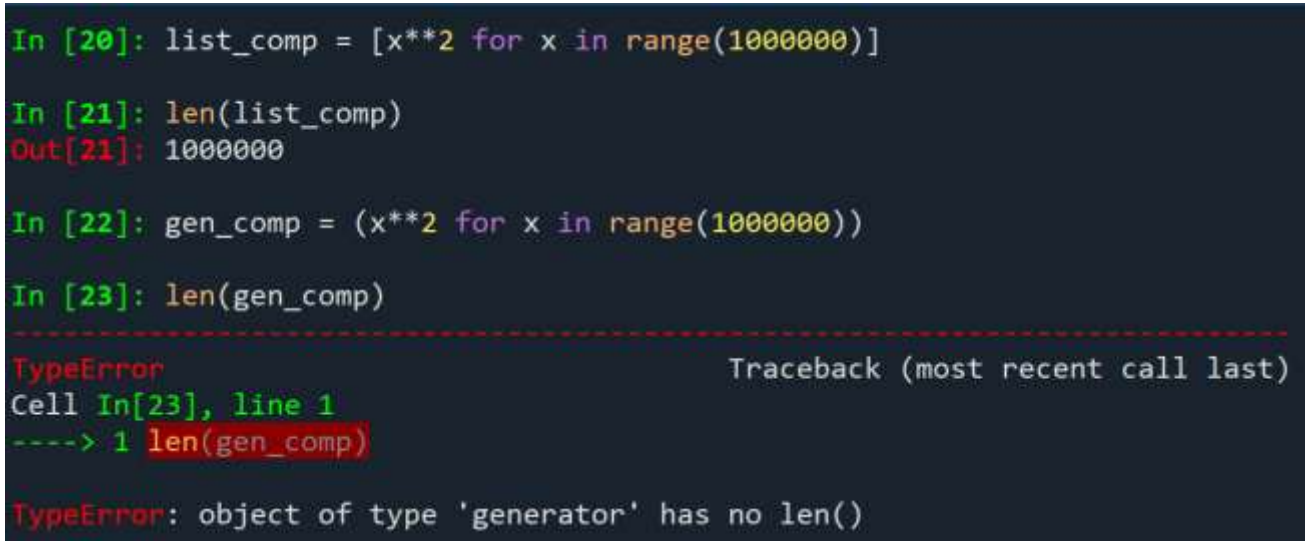
特性	生成器推导式	列表推导式	说明
内存占用	✅ 极低	❌ 高	生成器逐个产生元素，不占用大量内存
执行方式	✅ 惰性求值	❌ 立即执行	生成器需要时才计算，列表立即计算所有元素
大数据处理	✅ 适合	❌ 不适合	生成器可处理无限序列或超大数据集
性能	✅ 启动快	❌ 启动慢	生成器立即返回，列表需要先构建完整列表
复用性	❌ 一次性	✅ 可复用	生成器遍历一次后耗尽，列表可重复使用

3. 2. 3　生成器推导式

生成器推导式 vs 列表推导式

```
# 列表推导式 - 立即计算，占用内存
list_comp = [x**2 for x in range(1000000)]
# 立即创建100万个元素的列表
print(f"列表大小: {len(list_comp)}")

# 生成器推导式 - 惰性计算，节省内存
gen_comp = (x**2 for x in range(1000000))
# 不立即创建，只在需要时生成
print(f"生成器大小: 无法直接获取长度")
```



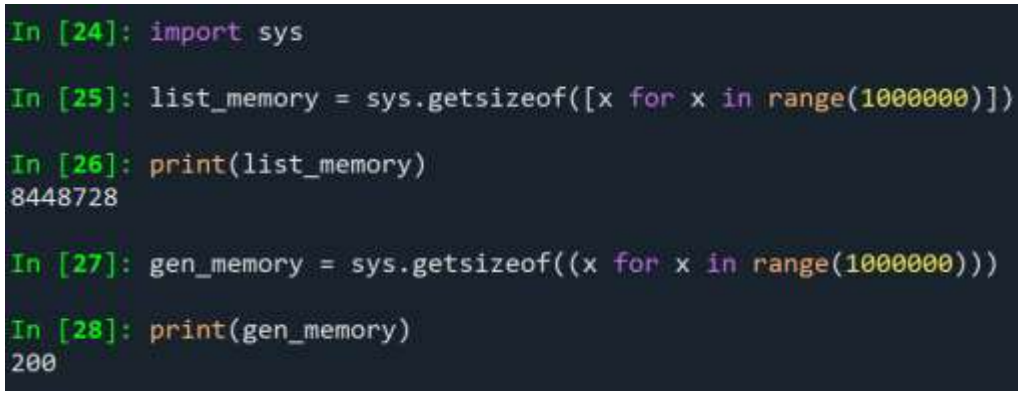
3. 2. 3　生成器推导式

生成器推导式 vs 列表推导式（内存效率对比）

```
import sys

# 列表推导式 - 占用大量内存
list_memory = sys.getsizeof([x for x in range(1000000)])
print(f"列表内存占用: {list_memory} bytes")

# 生成器推导式 - 占用极少内存
gen_memory = sys.getsizeof((x for x in range(1000000)))
print(f"生成器内存占用: {gen_memory} bytes")
```



3.2.3 生成器推导式

■生成器对象的作用：

- ✓ 高效率的遍历或查找序列的特殊形式
- ✓ 用作all(), any(), sorted()等内置函数的参数
- ✓ 赋值给列表的切片
- ✓ 其他可迭代对象出现的场合

■ 生成器对象支持的用法：

- ✓ 使用生成器对象的__next__()方法（了解）或者内置函数next()逐个获取元素
- ✓ 使用for循环来遍历
- ✓ 使用in运算符做查找

3.2.3 生成器推导式

■使用for循环直接遍历生成器对象中的元素

```
>>> g = ((i+2)**2 for i in range(10))
>>> for item in g:
        print(item, end=' ')
4 9 16 25 36 49 64 81 100 121

>>> for item in ((i+2)**2 for i in range(10)):  #更直接
        print(item, end=' ')

#实现对生成器的多次使用(了解，不常用)
>>> g = tuple(((i+2)**2 for i in range(10))) #转化为元组或列表 #内层()可去掉

>>> g = '((i+2)**2 for i in range(10))'  #把生成器转成字符串，每次使用调用eval()
>>> for item in eval(g):
       print(item, end=' ')
```

3.2.3 生成器推导式

```
>>> g = '((i+2)**2 for i in range(10))'
>>> a = [1,2,3]
>>> a[3:] = eval(g)    #把生成器赋值给列表的切片来替换列表中元素
>>> a
[1, 2, 3, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]

>>> sorted(eval(g), reverse=True)  #用作all(), any(), sorted()等内置函数的参数

[121, 100, 81, 64, 49, 36, 25, 16, 9, 4]

>>> all((i+2)**2 < 100 for i in range(10))  #生成器的()可省略
False
>>> all((i+2)**2 < 200 for i in range(10))
True
```

3.2.3 生成器推导式

■in运算：访问过的元素不再存在

```
>>> g = (i for i in range(10))
>>> 1 in g
True
>>> 2 in g
True
>>> 1 in g
False
>>> list(g)
[]
```

3. 2. 3　生成器推导式

```
# 验证生成器的惰性
>>> a = [1,2,3,4]
>>> g = (i**2 for i in a)
>>> a.clear()  #谨慎操作
>>> list(g)
[]
```

- 生成器推导式 (i**2 for i in a) 在创建时：
  - 不会立即执行计算
  - 不会复制列表 a 的数据
  - 只是保存了对列表 a 的引用
  - 实际计算推迟到迭代时进行
- 结果为空列表！ 因为当真正迭代生成器时，原列表 a 已经被清空了。

3.2.3 生成器推导式

■生成器对象的特性：

- 对生成器对象，没有任何方法可以再次访问已访问过的元素，也不支持使用下标访问其中的元素
- 当所有元素访问结束以后，生成器变“空”

3.2.3 生成器推导式

■使用内置函数next()依次获取生成器对象中的元素

```
>>> g = ((i+2)**2 for i in range(10))  #创建生成器对象
>>> type(g)
<class 'generator'>
>>> tuple(g)                          #将生成器对象转换为元组
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> list(g)                  #生成器对象已遍历结束，没有元素了
[]

>>> g = ((i+2)**2 for i in range(10))  #重新创建生成器对象
>>> next(g)                  #使用函数next()获取生成器对象中的元素
4
```

Practice

- 使用哪个内置函数来逐个访问生成器对象的元素？

✓ next()

- 下列语句输出？

```
>>> g = (i**2 for i in range(0,10,2))
>>> 10 in g
>>> list(g)
```

✓ False

✓ []　　#第二句in 操作会消耗元素：直到找到目标或确定不存在
　　#上一句访问10的时候已经便利一遍了

```
>>> g = (i**2 for i in range(0,10,2))
>>> 4 in g    #in 操作找到目标后立即停止，生成器指针会停留在找到位置的下一个元素之前
>>> list(g)   #list(g) 总是从当前指针位置开始收集剩余元素
```

✓ True

✓ [16, 36, 64]

习题

3.10 _____（可以、不可以）使用 del 命令来删除元组中的部分元素。

休息一下

子在川上曰：逝者如斯夫，不舍昼夜。

——《论语·子罕》

My mom always said life was like a box of chocolates. You never know what you're gonna get.
人生就像一盒巧克力，你永远不知道下一颗是什么味道.

——《阿甘正传》

第3章 详解Python序列结构

- 3.1字典：反映对应关系的映射类型
  - 3.3.1字典创建
  - 3.3.2字典元素的访问
  - 3.3.3元素的添加、修改与删除
  - 3.3.4标准库collections中与字典有关的类

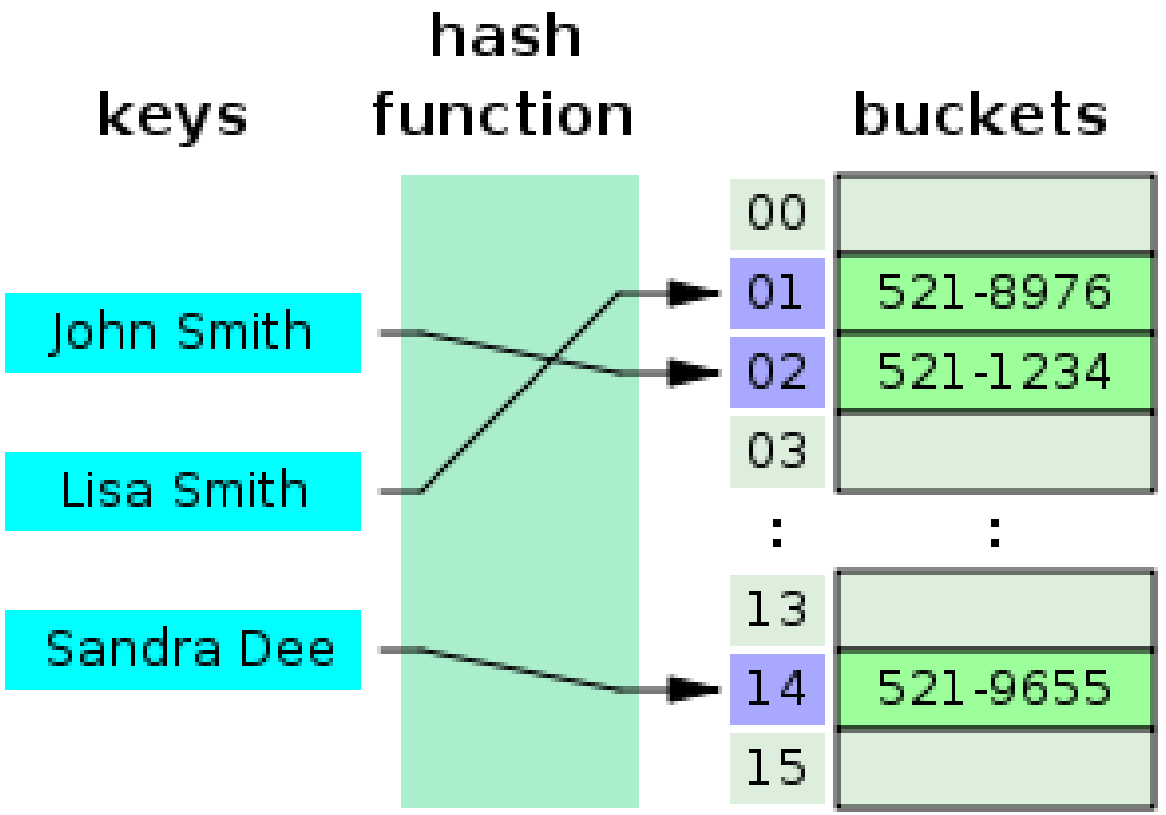
3.3 字典：反映对应关系的映射类型

```
>>> d = {'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> d = {('appphys', 1):'Xiong', ('jidiban', 1):'Xiong'} #元组可做键
```

- 字典（dictionary）是包含若干“键:值”元素的**无序**可变序列，表示一种映射关系
- 字典中元素的“键”必须是不可变（可哈希）数据类型，例如数字、字符串、元组等，但**不能**使用**列表**等可变类型**作为字典的“键”**
- 字典中的“键”不允许重复，而“值”是可以重复的：字典是一种离散函数

3.3 字典：反映对应关系的映射类型

- CPython对字典的实现（了解）：散列表（Hash table，也叫哈希表），是根据键（Key）而直接访问内存存储位置的数据结构



3.3 字典：反映对应关系的映射类型

■ Python字典一个应用（了解）：HTTP请求的“头”（header）--典型的网页爬虫基础模板

```
import requests

headers = {
'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.110 Safari/537.36',
'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8',
'Accept-Encoding': 'gzip, deflate',
'Accept-Language': 'zh-CN,zh;q=0.9,en;q=0.8,ja;q=0.7,zh-TW;q=0.6,th;q=0.5,ko;q=0.4,es;q=0.3',
}

url = r'https://book.douban.com/subject/30173345/'
r = requests.get(url, headers = headers)
print(r.text)
```

3.3 字典：反映对应关系的映射类型

■ Python字典另一个例子：摩尔斯电码

```
morse = {
"A" : ".-",
"B" : "-...",
"C" : "-.-.",
"D" : "-..",
"E" : ".",
"F" : "..-.",
"G" : "--.",
"H" : "....",
"I" : "..",
"J" : ".---",
"K" : "-.-",
"L" : ".-..",
"M" : "--",
"N" : "-.",
"O" : "---",
"P" : ".--.",
"Q" : "--.-",
"R" : ".-.",
"S" : "...",
"T" : "-",
"U" : "..-",
"V" : "...-",
"W" : ".--",
"X" : "-..-",
"Y" : "-.--",
"Z" : "--..",
"0" : "-----",
"1" : ".----",
"2" : "..---",
"3" : "...--",
"4" : "....-",
"5" : ".....",
"6" : "-....",
"7" : "--...",
"8" : "---..",
"9" : "----.",
"." : ".-.-.-",
"," : "--..--"
}
```

演示：

```
>>> len(morse)
38

>>> "a" in morse
False
>>> "A" in morse
True
>>> "a" not in morse
True

morse1 = dict(zip(morse.values(), morse.keys()))
```

第3章 详解Python序列结构

- 3.1字典：反映对应关系的映射类型
  - **3.3.1字典创建**
  - 3.3.2字典元素的访问
  - 3.3.3元素的添加、修改与删除
  - 3.3.4标准库collections中与字典有关的类

3. 3. 1　字典创建

■可以使用内置类dict以不同形式创建字典　#dir(dict)

```
>>> x = dict()                          #空字典
>>> type(x)
<class 'dict'>
>>> x = {}

>>> keys = ['a', 'b', 'c', 'd']
>>> values = [1, 2, 3, 4]
>>> dictionary = dict(zip(keys, values))     #根据已有数据创建字典
```

3. 3. 1　字典创建

```
>>> d = dict(name='Dong', age=39)          #以关键参数的形式创建字典
>>> d = {'name': 'Dong', 'age': 39}


>>> aDict = dict.fromkeys(['name', 'age', 'sex'])
                        #类方法的调用，不同于常见的实例方法
                        #以给定内容为“键”，创建“值”为空的字典
>>> aDict
{'age': None, 'name': None, 'sex': None}  #可用于初始化字典

>>> aDict = dict.fromkeys(['name', 'age', 'sex'],0)
                         #以给定内容为“键”，创建“值”为空的字典
>>> aDict
{'name': 0, 'age': 0, 'sex': 0}  #用于所有键需要相同的初始值的情况
```

3. 3. 1　字典创建

■使用字典对象的copy()方法获得字典的浅复制（了解）

```
>>> aDict = {'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> d = aDict.copy()
>>> d['score'][0] = 100
>>> aDict
{'age': 39, 'score': [100, 97], 'name': 'Dong', 'sex': 'male'}
                                #原理类似于列表中对可变元素的浅拷贝，本ppt32页
```

第3章 详解Python序列结构

- 3.1字典：反映对应关系的映射类型
  - 3.3.1字典创建
  - **3.3.2字典元素的访问**
  - 3.3.3元素的添加、修改与删除
  - 3.3.4标准库collections中与字典有关的类

3. 3. 2　字典元素的访问

■以“键”为下标访问对应的“值”
■如果字典中不存在这个“键”会抛出异常

```
>>> aDict = {'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict['age']                    #指定的“键”存在，返回对应的“值”
39


>>> 'address' in aDict
False

>>> aDict['address']                #指定的“键”不存在，抛出异常
KeyError: 'address'
```

3.3.2 字典元素的访问

- 字典对象的get(k[,d])方法用来返回指定“键”对应的“值”
- 允许指定该键不存在时返回特定的“值”

```
>>> aDict.get('age')                          #如果字典中存在该“键”则返回对应的“值”
39
>>> aDict.get('address', 'Not Exists.') #指定的“键”不存在时返回指定的默认值
'Not Exists.'
```

- (了解)字典对象的setdefault(k[,d])方法与get(k[,d])方法用法基本相同
- 但当键k不存在时，设置其对应的值为d

```
>>> aDict.setdefault('address', 'Not Exists.')
'Not Exists.'
>>> aDict
{'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male', 'address': 'Not Exists.'}
```

d[key]访问：用于“必须存在”场景，强调数据的完整性
get()访问：用于“可选存在”场景，提供安全访问
setdefault()访问：用于“初始化或获取”场景，简化代码

3. 3. 2　字典元素的访问

■对字典直接进行遍历时，默认是遍历字典的“键”

```
>>> aDict = {'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> for k in aDict:
        print(k, aDict[k])

age 39
score [98, 97]
name Dong
sex male
```

■遍历字典的“键值对”、“值”，必须使用字典对象的items()和values()方法

3. 3. 2　字典元素的访问

- 使用字典对象的items()方法可以返回字典的键、值对序列（可迭代的dict_items对象）

```
>>> aDict.items()
dict_items([('age', 39), ('score', [98, 97]), ('name', 'Dong'), ('sex', 'male'), ('address', 'Not Exists.')])        #(键,值)以元组形式输出
```

使用字典对象的values()方法可以返回字典的值序列

```
>>> aDict.values()
dict_values([39, [98, 97], 'Dong', 'male', 'Not Exists.'])
```

使用字典对象的keys()方法可以返回字典的键序列

```
>>> aDict.keys()
dict_keys(['age', 'score', 'name', 'sex', 'address'])
```

3. 3. 2　字典元素的访问

- **问题解决：**已知有一个包含一些同学成绩的字典，现在需要计算所有成绩的最高分、最低分、平均分，并查找所有最高分同学

```
>>> scores = {"Zhang San": 99, "Li Si": 78, "Wang Wu": 40, "Zhou Liu": 96,
              "Zhao Qi": 65, "Sun Ba": 90, "Zheng Jiu": 78, "Wu Shi": 99,
              "Dong Shiyi": 60}

>>> highest = max(scores.values())                    #最高分
>>> lowest = min(scores.values())                     #最低分
>>> average = sum(scores.values()) / len(scores)      #平均分
>>> highest, lowest, average
(99, 40, 72.33333333333333)

>>> highestPerson = [name for name, score in scores.items() if score == highest]
>>> highestPerson
['Wu Shi']
```

3. 3. 2　字典元素的访问

演示：

```
scores = {"Zhang San": 99, "Li Si": 78, "Wang Wu": 40, "Zhou Liu": 96,
          "Zhao Qi": 65, "Sun Ba": 90, "Zheng Jiu": 78, "Wu Shi": 99,
          "Dong Shiyi": 60}

print(list(scores.values()))
print(list(scores.items()))

highest = max(scores.values())
print(highest)

highestPerson = []
for name, score in scores.items():
    if score == highest:
        highestPerson.append(name)
    print(name, score, highestPerson)

print([name for name, score in scores.items() if score == highest])
```

3. 3. 2　字典元素的访问

- 对字典scores按姓名排序?

```
>>> sorted(scores.items())　#默认按键排序
```

- 对字典scores按分数排序？(了解)

```
>>> sorted(scores.items(), key=lambda v: v[1])
                                    #每个v是元组 (键，值)
```

第3章 详解Python序列结构

- 3.1字典：反映对应关系的映射类型
  - 3.3.1字典创建
  - 3.3.2字典元素的访问
  - **3.3.3元素的添加、修改与删除**
  - 3.3.4标准库collections中与字典有关的类

3.3.3 元素添加、修改与删除

■当以指定“键”为下标为字典元素赋值时，有两种含义

✓若该“键”存在，则表示修改该“键”对应的值

✓若不存在，则表示添加一个新的“键:值”对，也就是添加一个新元素

```
>>> aDict = {'age': 35, 'name': 'Dong', 'sex': 'male'}
>>> aDict['age'] = 39                    #修改元素值
>>> aDict
{'age': 39, 'name': 'Dong', 'sex': 'male'}

>>> aDict['address'] = 'SDIBT'   #山东工商学院      #添加新元素
>>> aDict
{'age': 35, 'name': 'Dong', 'sex': 'male', 'address': 'SDIBT'}
```

3.3.3 元素添加、修改与删除

- 使用字典对象的update()方法可以将另一个字典的“键:值”一次性全部添加到当前字典对象 #类似于列表中的extend()方法
- 如果两个字典中存在相同的“键”，则以新加字典中的“值”为准对当前字典进行更新

```
>>> aDict = {'age': 37, 'score': 98, 'name': 'Dong', 'sex': 'male'}
>>> aDict.update({'a':97, 'age':39})  #修改'age'键的值，同时添加新元素'a':97
>>> aDict
{'age': 39, 'score': 98, 'name': 'Dong', 'sex': 'male', 'a': 97}
#也可以是aDict.update(a=97, age=39)
#或者aDict.update([('a',97), ('age',39)])
```

3.3.3 元素添加、修改与删除

■如果需要删除字典中指定的元素，可以使用del命令 #列表中的del方法相同

```
>>> del aDict['age']                    #删除字典元素
>>> aDict
{'score': [98, 97], 'sex': 'male', 'a': 97, 'name': 'Dong'}
```

■也可以使用字典对象的pop()和popitem()方法弹出并删除指定的元素

```
>>> aDict = {'age': 37, 'score': 98, 'name': 'Dong', 'sex': 'male'}
>>> aDict.popitem()                     #弹出一个元素，对空字典会抛出异常
('age', 37)
>>> aDict.pop('sex')                    #弹出指定键对应的元素
'male'
>>> aDict
{'score': 98, 'name': 'Dong'}
```

■clear()可清空字典 #与列表中的clear方法相同

3. 3. 3　元素添加、修改与删除

- 问题解决：首先生成包含1000个随机字符的字符串，然后统计每个字符的出现次数

```
>>> import string
>>> import random
>>> x = string.ascii_letters + string.digits + string.punctuation
>>> y = [random.choice(x) for i in range(1000)]

>>> d = dict()                    #使用字典保存每个字符出现次数
>>> for ch in y:
        d[ch] = d.get(ch, 0) + 1    #否则返回0
                                    #ch第一次出现在d中会报错，因为d初始为[]，所以用get()
>>> print(d)
```

3.3.3 元素添加、修改与删除

- 问题解决：使用**字典推导式**快速创建字典(了解)
  {键表达式:值表达式 for value in seq if cond}

```
>>> import string
>>> import random
>>> x = string.ascii_letters + string.digits + string.punctuation
>>> y = [random.choice(x) for i in range(1000)]

>>> d = {ch:y.count(ch) for ch in x}
# 或 d = dict((ch, y.count(ch)) for ch in x)
>>> print(d)
```

- #查dir(dict)

Practice

- 如何创建空字典？

✓ `dict()`

✓ `{}`

- 如何以序列k的元素为键，以序列d的元素为值，创建字典？

✓ `dict(zip(k,d))`

- 分别使用哪种方法获取字典的键、值、键值对所构成的序列？

✓ `keys(), values(), items()`

- 如何以序列k的元素为键，以0为对应的值，创建字典？

✓ `dict(zip(k, (0 for i in range(len(k)))))`

✓ `{key:0 for key in k}`

✓ `dict.fromkeys(k, 0)`

习题

3.5 在 Python 中，字典和集合都是用一对_____作为界定符，字典的每个元素由两部分组成，即_____和_____，其中_____不允许重复。

3.6 使用字典对象的_____方法可以返回字典的“键:值”对，使用字典对象的_____方法可以返回字典的所有“键”，使用字典对象的_____方法可以返回字典的所有“值”。

3.7 假设有列表 a = ['name', 'age', 'sex'] 和 b = ['Dong', 38, 'Male']，请使用一条语句将这两个列表的内容转换为字典，并且以列表 a 中的元素为“键”，以列表 b 中的元素为“值”，这个语句可以写为_____。

第3章 详解Python序列结构

- 3.1字典：反映对应关系的映射类型
  - 3.3.1字典创建
  - 3.3.2字典元素的访问
  - 3.3.3元素的添加、修改与删除
  - **3.3.4标准库collections中与字典有关的类**

3.3.4 标准库collections中与字典有关的类

■collections模块的Counter类：频次统计
■提供了更多的功能，例如查找出现次数最多的元素

```
>>> import string
>>> import random
>>> x = string.ascii_letters + string.digits + string.punctuation
>>> y = [random.choice(x) for i in range(1000)]

>>> from collections import Counter
>>> frequences = Counter(y)
>>> frequences.items()
>>> frequences.most_common(1)        #返回出现次数最多的1个字符及其频率
>>> frequences.most_common(3)        #返回出现次数最多的前3个字符及其频率
```

3. 3　字典总结

1. 创建字典

方法	语法	示例	说明
直接创建	{key: value}	d = {'a': 1, 'b': 2}	最常用
dict() 函数	dict()	d = dict(a=1, b=2)	键必须是合法标识符
键值对序列	dict([(k,v)])	d = dict([('a',1),('b',2)])	从元组列表创建
		或 d = dict(zip(['a','b'],[1,2]))	利用zip函数
fromkeys()	dict.fromkeys(seq, val)	d = dict.fromkeys(['a','b'], 0)	所有键相同值
copy()	d.copy()	a = d.copy	浅复制
字典推导式	{k:v for}	d = {x: x**2 for x in range(3)}	动态创建
		或 d = dict(x: x**2 for x in range(3))	

3. 3　字典总结

2. 访问元素

方法	语法	示例	返回值	键不存在时
键访问	d[key]	d['a']	值	KeyError
get()	d.get(key, default)	d.get('a', 0)	值或default	返回default
setdefault()	d.setdefault(key, default)	d.setdefault('a', 0)	值或设置default	设置并返回default

3. 查询信息

方法	语法	示例	返回值	说明
in / not in	key in d	'a' in d	bool	键是否存在
len()	len(d)	len(d)	int	键值对数量
keys()	d.keys()	d.keys()	键视图	所有键
values()	d.values()	d.values()	值视图	所有值
items()	d.items()	d.items()	键值对视图	所有键值对

3.3 字典总结

4. 添加或修改字典

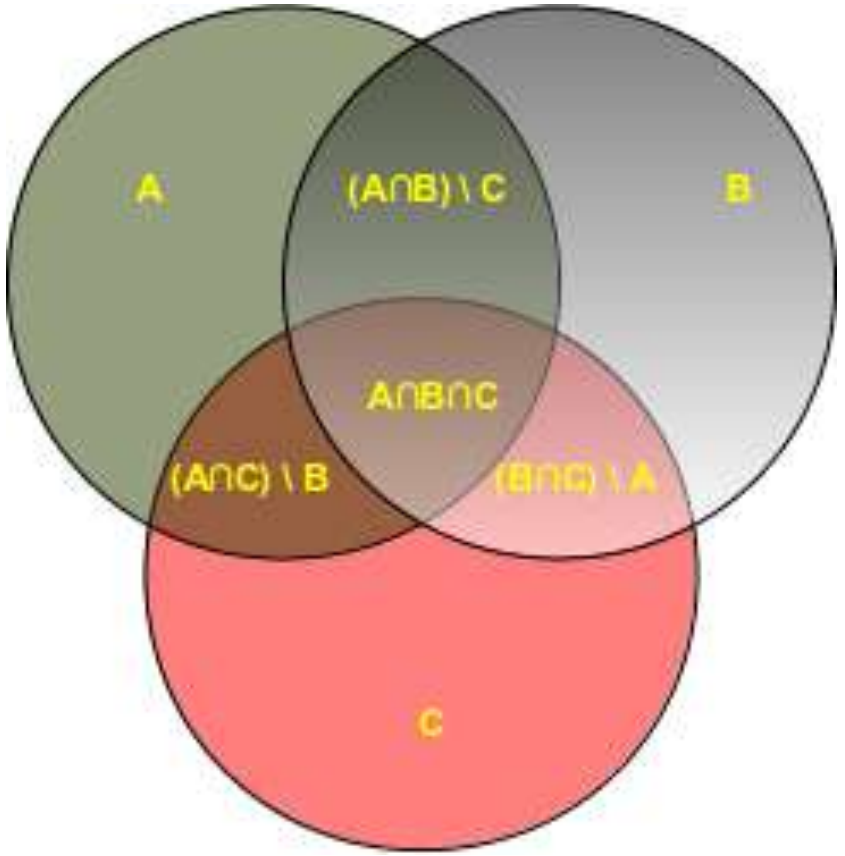
方法	语法	示例	说明
直接赋值	d[key] = value	d['a'] = 1	添加或修改
update()	d.update(other)	d.update({'c':3})	合并字典
	d.update(key=value)	d.update(c=3, d=4)	关键字参数
	d.update([(k,v)])	d.update([('c',3)])	键值对序列

5. 删除元素

方法	语法	示例	返回值	键不存在时
del	del d[key]	del d['a']	None	KeyError
pop()	d.pop(key, default)	d.pop('a')	删除的值	KeyError或缺省值
popitem()	d.popitem()	d.popitem()	(key, value)	KeyError(空字典)
clear()	d.clear()	d.clear()	None	清空所有元素

第3章 详解Python序列结构

- 3.4集合：元素之间不允许重复
  - 3.4.1集合对象的创建
  - 3.4.2集合操作与运算
  - 3.4.3集合应用案例
- 3.5序列解包的多种形式和用法



3.4 集合

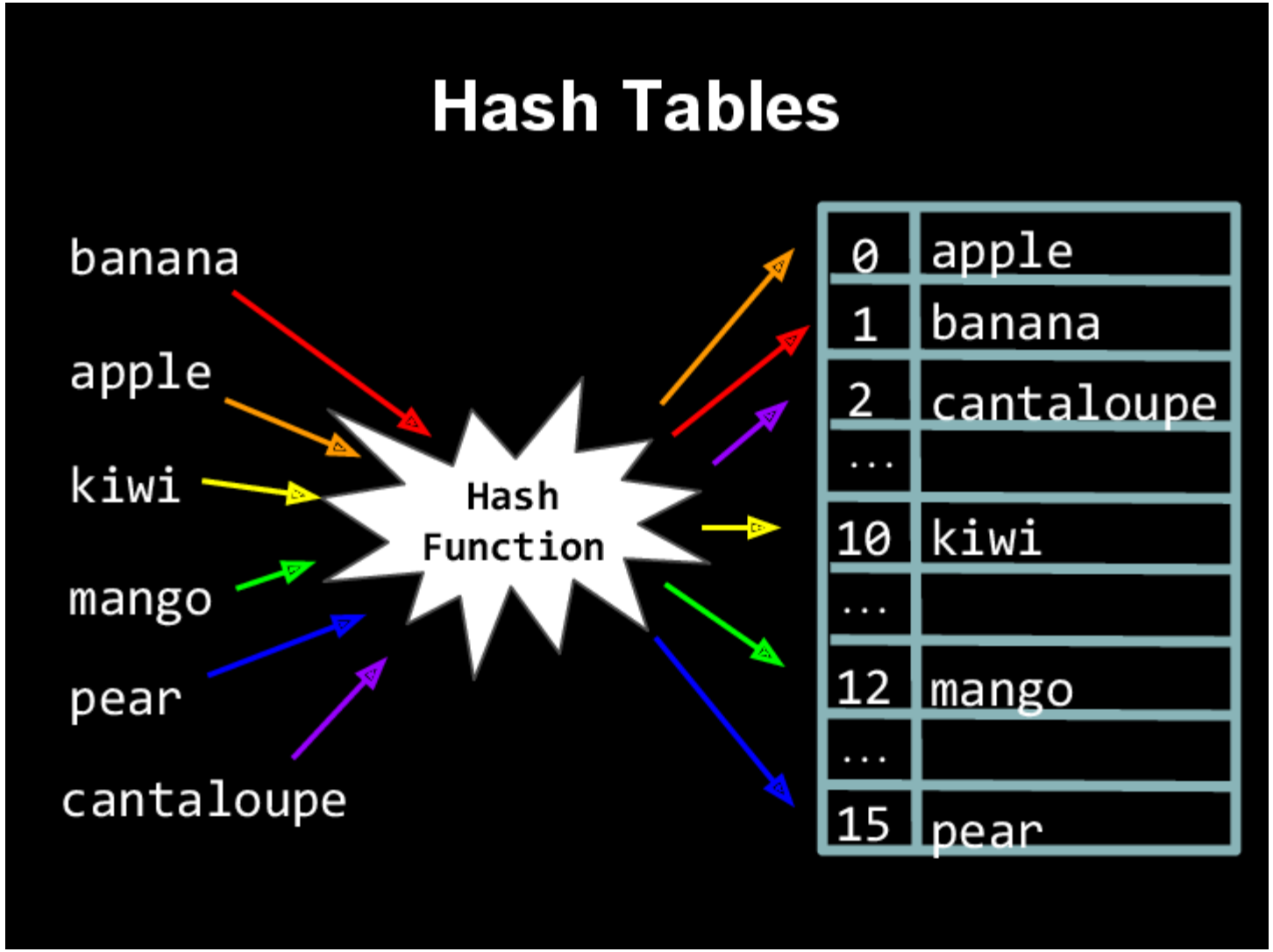
```
>>> fruits = {'banana', 'apple', 'pear'}
```

- 集合（set）属于Python**无序可变**序列
- 一个集合内的每个元素都是唯一的，元素之间不允许重复
- 集合中只能包含数字、字符串、元组等不可变类型（或者说可哈希）的数据，而不能包含列表、字典、集合等可变类型的数据

第3章 详解Python序列结构

- 3.4集合：元素之间不允许重复
  - **3.4.1集合对象的创建**
  - 3.4.2集合操作与运算
  - 3.4.3集合应用案例
- 3.5序列解包的多种形式和用法

3.4 集合



集合底层是字典实现的，

集合的所有元素都是字典的”键”，

因此是不能重复且唯一的。

3.4.1 集合对象的创建

■直接将集合赋值给变量即可创建一个集合对象

```
>>> a = {3, 5}                    #创建集合对象
>>> type(a)                       #查看对象类型
<class 'set'>
```

■可以使用函数set()函数将**列表**、元组、字符串、range对象等其他可迭代对象转换为集合

- 如果原序列中存在重复元素，则在转换为集合的时候只保留一个
- 如果原序列中有不可哈希的值，无法转换成为集合，抛出异常

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

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

Build an unordered collection of unique elements.
```

```
In [4]: set(0, 1, 2, 3, 0, 1, 2, 3, 7, 8)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 set(0, 1, 2, 3, 0, 1, 2, 3, 7, 8)

TypeError: set expected at most 1 argument, got 10
```

3. 4. 1　集合对象的创建

```
>>> a_set = set(range(8, 14))                    #把range对象转换为集合
>>> a_set
{8, 9, 10, 11, 12, 13}

>>> b_set = set([0, 1, 2, 3, 0.0, 1.0, 2, 3, 7, 8])   #转换时自动去掉重复元素
>>> b_set
{0, 1, 2, 3, 7, 8}            # 0 == 0.0，整数会被提升为浮点数，再进行比较

>>> x = set()                                  #空集合  #{}代表空字典

>>> set('aaabbccdd')
{'a', 'b', 'c', 'd'}

  (了解)
>>> set([[1,2],[3,4]])  #TypeError: unhashable type: 'list'
>>> set([{1,2},{3,4}])  #TypeError: unhashable type: 'set'
>>> set([{'a':1},{'b':2}])  #TypeError: unhashable type: 'dict'
>>> set([(1,2),(3,4)])   #{(1, 2), (3, 4)}
```

3. 4. 1　集合对象的创建

■使用集合推导式快速生成集合；也可以自动去重（了解）

{表达式 for v in seq if cond}

```
>>> set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

>>> {i**2%5 for i in range(10)}          #自动去重
{0, 1, 4}

>>> set(i**2%5 for i in range(10))  #{}换为set()也可以
{0, 1, 4}
```

第3章 详解Python序列结构

- 3.4集合：元素之间不允许重复
  - 3.4.1集合对象的创建
  - **3.4.2集合操作与运算**
  - 3.4.3集合应用案例
- 3.5序列解包的多种形式和用法

3. 4. 2　集合操作与运算

（1）集合元素增加与删除

- add()方法可以增加**新元素**，如果该元素已存在则忽略该操作，不会抛出异常
  #类似于列表中的append()
- update()方法用于**合并**另外一个**集合**中的元素到当前集合中，并自动去除重复元素
- add()和update()均为**原地**操作，并集运算符|生成**新**集合

```
>>> s = {1, 2, 3}
>>> s.add(3)                    #添加元素，重复元素自动忽略
>>> s
{1, 2, 3}

>>> s.update({3,4})             #更新当前字典，自动忽略重复的元素
>>> s
{1, 2, 3, 4}
```

3. 4. 2　集合操作与运算

- pop()方法用于**随机**删除并返回集合中的一个**元素**，如果集合为空则抛出异常
- remove()方法用于删除集合中的**元素**，如果指定元素不存在则**抛出异常**(严肃)
- discard()用于从集合中删除一个**特定**元素，如果元素不在集合中则忽略该操作
- clear()方法清空集合删除所有元素

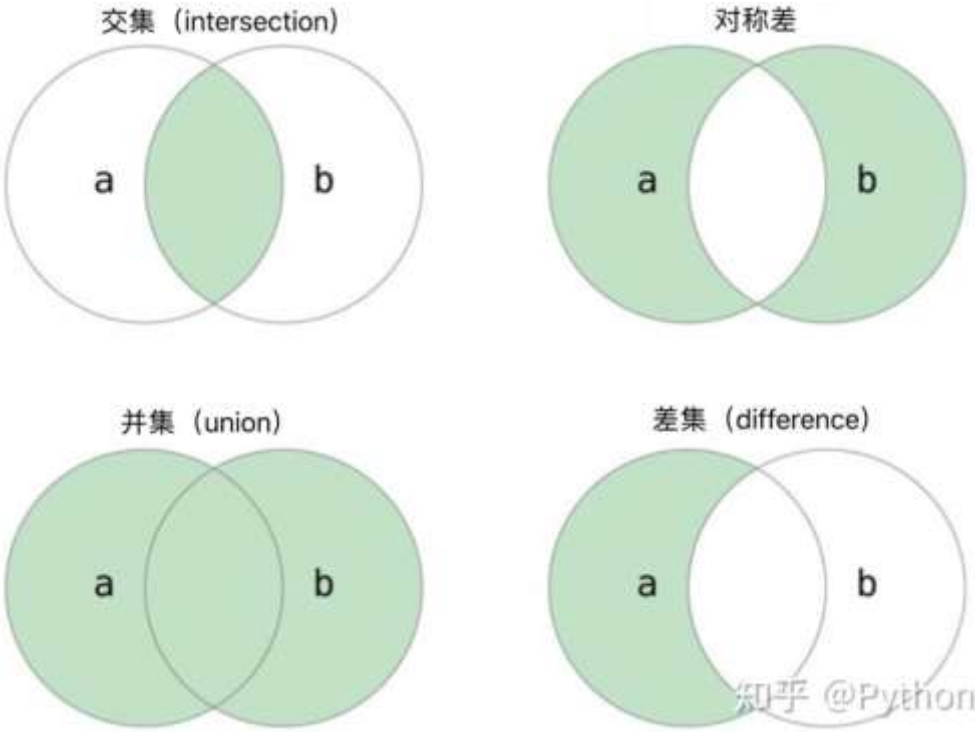
```
>>> s.discard(5)        #删除元素，不存在则忽略该操作
>>> s
{1, 2, 3, 4}
>>> s.remove(5)         #删除元素，不存在就抛出异常
KeyError: 5
>>> s.pop()             #删除并返回一个元素
1
>>> s.clear()
```

•理解与记忆：
•pop(弹出)：像从袋子里随机拿出一个东西（不知道是哪一个）。
•remove(移除)：你要移除一个具体的东西，如果找不到，你会感到异常（报错）。
•　　态度很严肃！！！
•discard(丢弃)：你要丢弃一个东西，如果找不到，那就算了（不报错）。
•　　态度很随意！！！

3.4.2 集合操作与运算

- （复习）集合的交集、并集、对称差集等运算符与位运算符相同，差集使用减号运算符（注意，并集运算符不是加号）。

```
>>> {1, 2, 3} | {3, 4, 5}          #并集，自动去除重复元素
{1, 2, 3, 4, 5}
>>> {1, 2, 3} & {3, 4, 5}          #交集
{3}
>>> {1, 2, 3} ^ {3, 4, 5}          #对称差集
{1, 2, 4, 5}
>>> {1, 2, 3} - {3, 4, 5}          #差集
{1, 2}
```



3. 4. 2　集合操作与运算

（2）集合运算

```
>>> a_set = {8, 9, 10, 11, 12, 13}
>>> b_set = {0, 1, 2, 3, 7, 8}

>>> a_set | b_set                          #并集
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set.union(b_set)                     #并集（了解）
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}

>>> a_set & b_set                          #交集
{8}
>>> a_set.intersection(b_set)              #交集（了解）
{8}
```

3.4.2 集合操作与运算

```
>>> a_set.difference(b_set)                #差集（了解）
{9, 10, 11, 12, 13}
>>> a_set - b_set
{9, 10, 11, 12, 13}

>>> a_set.symmetric_difference(b_set) #对称差集（了解）
{0, 1, 2, 3, 7, 9, 10, 11, 12, 13}
>>> a_set ^ b_set
{0, 1, 2, 3, 7, 9, 10, 11, 12, 13}
>>> (a_set|b_set) - (a_set&b_set)
```

3.4.2 集合操作与运算

```
>>> x = {1, 2, 3}
>>> y = {1, 2, 5}
>>> z = {1, 2, 3, 4}

>>> x < y                                  #比较集合包含关系
False
>>> x < z                                  #真子集
True
>>> y < z
False

>>> {1, 2, 3} <= {1, 2, 3}                 #子集
True
```

3.4 集合总结

基本方法

方法	语法	描述	返回值	元素不存在时	示例
add()	set.add(element)	添加元素到集合	None	-	s.add(5)
update()	set.update(iterable)	批量添加元素	None	-	s.update([1,2,3])
pop()	set.pop()	随机移除并返回元素	被移除的元素	集合为空时报错	x = s.pop()
remove()	set.remove(element)	移除指定元素	None	报错(KeyError)	s.remove(5)
discard()	set.discard(element)	安全移除元素	None	静默忽略	s.discard(5)
clear()	set.clear()	清空所有元素	None	-	

3.4 集合总结

查询操作

方法	语法	描述	返回值	示例
in	element in set	检查元素是否存在	bool	5 in s
not in	element not in set	检查元素是否不存在	bool	5 not in s
len()	len(set)	返回元素个数	int	len(s)

集合运算

方法	语法	描述	数学符号	示例
union()	set1.union(set2)	并集	A ∪ B	s1.union(s2)
intersection()	set1.intersection(set2)	交集	A ∩ B	s1.intersection(s2)
difference()	set1.difference(set2)	差集	A - B	s1.difference(s2)
symmetric_difference()	set1.symmetric_difference(set2)	对称差集	A Δ B	s1.symmetric_difference(s2)

3.4 集合总结

集合关系判断（了解）

方法	语法	描述	返回值	示例
issubset()	set1.issubset(set2)	子集判断	bool	s1.issubset(s2)
issuperset()	set1.issuperset(set2)	超集判断	bool	s1.issuperset(s2)
isdisjoint()	set1.isdisjoint(set2)	是否无交集	bool	s1.isdisjoint(s2)

修改原集合的方法（了解）

方法	语法	描述	示例
intersection_update()	set1.intersection_update(set2)	交集并更新	s1.intersection_update(s2)
difference_update()	set1.difference_update(set2)	差集并更新	s1.difference_update(s2)
symmetric_difference_update()	set1.symmetric_difference_update(set2)	对称差集并更新	s1.symmetric_difference_update(s2)

3.4 集合总结

运算符版本

方法	运算符	描述	示例
并集	|	全部元素去重	集合合并
交集	&	共同元素	s1 & s2
差集	-	只在第一个集合中的元素	s1 - s2
对称差集	^	不同时在两个集合中的元素	s1 ^ s2
子集	<=	子集判断	s1 <= s2
真子集	<	真子集判断	s1 < s2
超集	>=	超集判断	s1 >= s2
真超集	>	真超集判断	s1 > s2

3. 4　集合总结

集合同时提供方法和运算符两种版本：

主要区别和适用场景

方面	方法版本	运算符版本
可读性	语义明确，像英语句子	简洁，数学感强
灵活性	可接受任意可迭代对象	只能用于集合类型
链式调用	支持链式调用	不支持链式调用
多个操作	可处理多个集合	只能两个集合操作

```
# 方法版本更灵活 - 接受任何可迭代对象
set_a = {1, 2, 3}
result1 = set_a.union([3, 4, 5])      # 列表
result2 = set_a.intersection(range(5)) # 范围对象
result3 = set_a | [3, 4, 5]         # 错误!
```

3. 4　集合总结

```
set_a = {1, 2}
set_b = {2, 3}
set_c = {3, 4}
set_d = {4, 5}

# 方法版本支持链式调用
result = (set_a.union(set_b).intersection(set_c).difference(set_d))

# 运算符版本不支持优雅的链式
result = ((set_a | set_b) & set_c) - set_d  # 括号嵌套，可读性差


# 方法版本 - 支持多个集合
union_all = set_a.union(set_b, set_c)      # {1, 2, 3, 4}
# 运算符版本 - 只能两个集合
union_ab = set_a | set_b | set_c           # 需要多次操作
```

第3章 详解Python序列结构

- 3.4集合：元素之间不允许重复
  - 3.4.1集合对象的创建
  - 3.4.2集合操作与运算
  - **3.4.3集合应用案例**
- 3.5序列解包的多种形式和用法

3.4.3 集合应用案例

- 可以使用集合快速提取序列中单一元素，即提取出序列中所有不重复元素。如果使用传统方式的话，需要编写下面的代码：

```
>>> import random
#生成100个介于0到9999之间的随机数
>>> listRandom = [random.choice(range(10000)) for i in range(100)]
          #or listRandom = random.choices(range(10000), k=100)
          #or listRandom = [random.randint(0, 9999) for _ in range(100)]
>>> noRepeat = []
>>> for i in listRandom :
        if i not in noRepeat :
           noRepeat.append(i)
```

- 使用集合只需要下面这么一行代码

```
>>> newSet = set(listRandom)
```

3.4.3 集合应用案例

- **问题解决：**返回指定范围内一定数量的不重复数字。

```
import random

def randomNumbers(number, start, end):
    '''使用集合来生成number个介于start和end之间的不重复随机整数'''
    data = set()
    while len(data)<number: #当len(data)=number-1时，add最后一个数
        element = random.randint(start, end) #可能会产生重复数字
        data.add(element)    #自动去重
    return data
```

3.4.3 集合应用案例

```
import random  #更高效

def randomNumbers(number, start, end):
    '''使用集合来生成number个介于start和end之间的不重复随机整数'''
    data = set()
    all_data = set(range(start, end+1)) #所有可能数字
    for _ in range(number):
        if all_data-data:
            element = random.choice(tuple(all_data-data)) #choice(seq)
                #直接从剩余数字中选择，避免重复尝试已选中的数字，保证100%效率
            data.add(element)
    return data
```

```
import itertools
lst = list(itertools.combinations(range(10),3))  #给出所有可能的选择
                                                 #可用import random；random.choice(lst)随机选一个
for i,j,k in lst:                                #或用import random；random.choices(lst,k=N)随机选N个
    print(i,j,k)
```

3.4.3 集合应用案例

- **问题解决：**下面两段代码用来测试指定列表中是否包含非法（不规范）数据，很明显第二段使用集合的代码更高效一些。

```
import random

lstColor = ('red', 'green', 'blue')
colors = [random.choice(lstColor) for i in range(10000)]
colors[9000]='white' #给一个非法数据来测试

#测试1
for item in colors:                                    #遍历列表中的元素并逐个判断
    if item not in lstColor:
        print('error:', item)
        break
#测试2
if (set(colors)-set(lstColor)):#set(colors)之后自动去重，应该和set(lstCorlor)相同
    print('error')                          #即两个集合差集应为空，如不为空，则存在不合法数据
```

3. 4. 3　集合应用案例

- **问题解决：**假设已有若干用户名字及其喜欢的电影清单，现有某用户，已看过并喜欢一些电影，现在想找个新电影看看，又不知道看什么好。
- 思路：根据已有数据，查找与该用户爱好最相似的用户，也就是看过并喜欢的电影与该用户最接近，然后从那个用户喜欢的电影中选取一个当前用户还没看过的电影，进行推荐。

- 演示：豆瓣　#https://movie.douban.com/
- **数据才是关键！**　#大数据时代，搜集数据比处理数据更重要

3.4.3 集合应用案例

```
from random import randrange

# 其他用户喜欢看的电影清单
data = {'user'+str(i):{'film'+str(randrange(1, 10))  #字典推导式
                       for j in range(randrange(15))} #集合存电影清单，不重复
        for i in range(10)}    #10个用户,10部电影
                    #15 是一个上限值，控制每个用户观看电影数量的随机范围，提高命中率
print('历史数据: ')
for u, f in data.items():
    print(u, f, sep=':')
```

3.4.3 集合应用案例

```
# 待测用户曾经看过并感觉不错的电影
user = {'film1', 'film2', 'film3'}

#待测用户与其他用户喜欢看的电影的交集的大小
[ (k, len(value & user)) for k, value in data.items() ] #列表推导式 #此句可不要

# 查找与待测用户最相似的用户和Ta喜欢看的电影
similarUser, films = max(data.items(), key=lambda item: len(item[1]&user))
                                                    #item[1]是电影集合，即values
print('和您最相似的用户是：', similarUser)
print('Ta最喜欢看的电影是：', films)
print('Ta看过的电影中您还没看过的有：', films-user)
```

Practice

- 如何创建空集合？
✓ set()

- 使用哪两个方法为集合增加元素？
✓ add(), update()

- 使用哪两个方法删除集合中的特定元素，这两个方法有什么区别？
✓ remove(), discard()
✓ discard()方法当元素不存在时不会抛出异常，remove()反之

- 集合的^运算符做什么运算？ {1,2,3,4} ^ {2,3,5}的值是？
✓ 对称差集
✓ {1,4,5}

习题

3.13 表达式 {1, 2, 3} < {3, 4, 5} 的值为_____。

3.16 单选题：表达式 set().union([1,2,3], (4,5), {6,7}) 的值为（　　）。

A. {1, 2, 3, 4, 5, 6, 7}　　B. {1, 2, 3, 4, 5}

C. {4, 5, 6, 7}　　D. {4, 5}

3.17 单选题：已知 x = {1: 'a', 2: 'b', 3: 'c'} 和 y = {1, 3, 4}，那么表达式 x.keys() - y 的值为（　　）。

A. {2}　　B. {3}

C. {1, 3}　　D. 表达式错误，无法计算

3.18 单选题：已知 x = {1: 3, 2: 1, 3: 1} 和 y = {1, 3, 4}，那么表达式 x.values() - y 的值为（　　）。

A. {2}　　B. {3}

C. {1, 3}　　D. set()

习题

3.26　多选题：下面可以使用表示位置或序号的整数做下标访问其中元素的有（　　）。

A．列表　　B．元组　　C．字典　　D．集合

E．字符串

3.27　多选题：下面可以支持下标运算的有（　　）。

A．列表　　B．元组　　C．字典　　D．集合

E．字符串

总结：Python序列结构

1. 基本特性对比

特性	List（列表）	Tuple（元组）	Dict（字典）	Set（集合）
可变性	✅ 可变	❌ **不可变**	✅ 可变	✅ 可变
有序性	✅ 有序	✅ 有序	❌ **无序**	❌ **无序**
语法	[]	()	{key: value}	{} 或 set()
元素要求	任意类型	任意类型	**键：唯一且不可变**	**唯一且不可变**
空对象创建	list() 或 []	tuple() 或 ()	dict() 或 {}	set()

总结：Python序列结构

2. 创建方法

操作	List	Tuple	Dict	Set
空对象	lst = [] 或 list()	tup = () 或 tuple()	d = **{}** 或 dict()	s = set(), **无{}**
带值创建	[1, 2, 'a']	(1, 2, 'a')	{'a': 1, 'b': 2}	{1, 2, 3}
单元素	[1]	**(1,)（必须加逗号）**	{'key': value}	{element}
从可迭代对象	list('abc')	tuple([1, 2, 3])	dict([('a',1)])	set([1,2,2,3])

总结：Python序列结构

3. 增加/修改操作

操作	List	Tuple	Dict	Set
添加单个	append(element)	❌ 不支持	**d[key] = value**	add(element)
添加多个	extend(iterable)	❌ 不支持	update(dict)	update(iterable)
插入指定位置	insert(index, element)	❌ 不支持	❌ 不支持	❌ 不支持
修改元素	lst[index] = new_value	❌ 不支持	d[key] = new_value	❌ 不支持

总结：Python序列结构

4. 删除元素

集合专注于成员存在与否的测试，提供安全discard()和严格remove()两种删除方式

操作	List	Tuple	Dict	Set
按**值**删除	remove(element)	❌ 不支持	❌ **不支持**	remove(element) 或 **discard**(element)
按**位置/键**删除	pop(index)	❌ 不支持	pop(key)	❌ 不支持
删除末尾/随机	pop()（末尾）	❌ 不支持	popitem()	**pop()（随机）**
清空所有	clear()	❌ **不支持**	clear()	clear()
del语句	del lst[index]	❌ **不支持删除元素** 只可删除标签del t	del d[key]	❌ **不支持删除元素** 只可删除标签del s

总结：Python序列结构

5. 访问与查询

d[key]访问：用于“必须存在”场景，强调数据的完整性
get()访问：用于“可选存在”场景，提供安全访问
setdefault()访问：用于“初始化或获取”场景，简化代码

操作	List	Tuple	Dict	Set
按**索引/键**	lst[index]	tup[index]	d[key] 或 **d.get(key)** 或 **d.setdefault(key)**	❌ **不支持**
切片操作	lst[start:end]	tup[start:end]	❌ 不支持	❌ 不支持
检查存在	element in lst	element in tup	key in d	element in s
长度	len(lst)	len(tup)	len(d)	len(s)

总结：Python序列结构

6. 复制

方法	List	Tuple	Dict	Set
赋值	new = original	new = original **new = original[:]** **new = tuple(original)**	new = original	new = original
浅拷贝	new = original.copy() new = original[:] new = list(original)		new = original.copy() new = dict(original)	new = original.copy() new = set(original)

操作类型	描述	内存关系	修改影响
赋值	创建对象的**引用**（别名）	共享同一内存地址	**完全同步**，修改任一都会影响另一个
浅拷贝	创建新对象，复制第一层元素	不同内存地址，**但嵌套的可变对象共享**	修改第一层元素互不影响，**修改嵌套的可变对象会相互影响**

总结：Python序列结构

7. 其他操作与方法

操作	List	Tuple	Dict	Set
排序	**sort()**	❌ 不支持	❌ 不支持	❌ 不支持
反转	**reverse()**	❌ 不支持	❌ 不支持	❌ 不支持
计数	count(element)	count(element)	❌ 不支持	❌ 不支持
索引查找	index(element)	index(element)	❌ 不支持	❌ 不支持
集合运算	❌ 不支持	❌ 不支持	❌ 不支持	**并集(|), 交集(&), 差集(-)等**
获取所有键	❌ 不支持	❌ 不支持	**keys()**	❌ 不支持
获取所有值	❌ 不支持	❌ 不支持	**values()**	❌ 不支持

总结：Python序列结构

8. 核心用途

- List：**有序**序列，频繁增删改
- Tuple：**不可变**数据，安全可靠
- Dict：键值映射，快速**查找**
- Set：**去重**存储，集合运算

第3章 详解Python序列结构

- 3.4集合：元素之间不允许重复
  - 3.4.1集合对象的创建
  - 3.4.2集合操作与运算
  - 3.4.3集合应用案例
- **3.5序列解包的多种形式和用法**

3.5 序列解包的多种形式和用法

序列解包（Sequence Unpacking）是指将一个序列（如列表、元组、字符串等）或可迭代对象的元素分解并赋值给多个变量的操作。

1. 列表（List）

```
>>> my_list = [1, 2, 3]
>>> a, b, c = my_list
>>> a, b, c      # 输出: (1, 2, 3)
>>> a            # 输出: 1
>>> b            # 输出: 2
>>> c            # 输出: 3
```

2. 元组（Tuple）

```
>>> my_tuple = (4, 5, 6)
>>> x, y, z = my_tuple
>>> x, y, z  # 输出: (4, 5, 6)
```

3. 5　序列解包的多种形式和用法

3. 字符串（String）

```
>>> my_str = "abc"
>>> x, y, z = my_str
>>> x, y, z  # 输出: ('a', 'b', 'c')
```

4. 字典（Dictionary）

```
>>> my_dict = {'a': 1, 'b': 2, 'c': 3}
>>> k1, k2, k3 = my_dict  # 默认解包键
>>> k1, k2, k3   # 输出:  ('a', 'b', 'c')
>>>for key, value in my_dict.items():
        print(key, value)  # 输出: a 1
                                  b 2
                                  c 3  #解包键值对
```

3. 5　序列解包的多种形式和用法

5. 集合（Set）

注意：集合是无序的，解包顺序不确定。#不建议用

```
>>> my_set = {7, 8, 9}
>>> a, b, c = my_set
>>> a, b, c                    # 顺序可能随机，如(8, 9, 7)
```

6. 生成器（Generator）

```
>>> gen = (x**2 for x in range(3))
>>> u, v, w = gen
>>> u, v, w   # 输出: (0, 1, 4)
```

3. 5　序列解包的多种形式和用法

7. range对象

```
>>> r = range(3)
>>> i, j, k = r
>>> i, j, k  # 输出: (0, 1, 2)
```

8. 嵌套序列

```
>>> data = [1, (2, 3), 4]
>>> a, (b, c), d = data
>>> a, b, c, d  # 输出: (1, 2, 3, 4)
```

3.5 序列解包的多种形式和用法

9. 使用星号（*）处理不定长元素

```
>>>numbers = [1, 2, 3, 4, 5]
>>>first, *middle, last = numbers
>>>first      # 输出: 1
>>>middle  # 输出: [2, 3, 4]
>>>last       # 输出: 5
```

10. map对象

```
>>>  x, y, z = map(str, range(3))
>>> x, y, z   #输出: ('0', '1', '2')
```

11. sorted对象

```
>>>  x, y, z = sorted([1, 3, 2])
>>> x, y, z   #输出: (1, 2, 3)
```

3.5 序列解包的多种形式和用法

12. 以两个循环变量遍历序列中的元组的形式也称为序列解包

```
>>> keys = ['a', 'b', 'c', 'd']
>>> values = [1, 2, 3, 4]
>>> for k, v in zip(keys, values):
        print(k, v)

#输出:
a 1
b 2
c 3
d 4

>>> for i, v in enumerate(keys):
        print(i, v)

#输出:
0 a
1 b
2 c
3 d

>>> s = {'a':1, 'b':2, 'c':3}
>>> for k, v in s.items():
    print(k, v)

#输出:
a 1
b 2
c 3
```

3.5 序列解包的多种形式和用法

注意事项： 元素数量匹配

```
>>> x, y = [1, 2, 3]  # ValueError: too many values to unpack
```

总结：

- ✓ 任何可迭代对象都支持序列解包，包括但不限于列表、元组、字符串、字典、集合、生成器等。
- ✓ 解包时需注意结构匹配和元素数量的一致性。

3.5 序列解包的多种形式和用法

重要应用：实现两个变量的交换

```
>>> x = [1,2]
>>> y = [3,4,5]
>>> x, y = y, x      #交换两个变量的值
                     #本质 (x, y) = (y, x) 或 (x, y) = [y, x]
>>> x
[3,4,5]
>>> y
[1,2]
```

作业4.3.3

1.创建一个字典d，键为26个小写英文字母，值为区间[0,10]中的随机整数

2.print字典d的值的最大值、最小值、平均值

3.使用列表推导式生成在[6,9]区间内的值所对应的键组成的列表

4.设a和b是两个列表，利用集合运算写表达式判断(1)a的所有元素都属于b，（2）a的所有元素都不属于b，（3）a有至少一个元素属于b