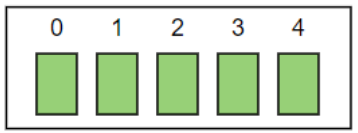
字典和集合

主要内容

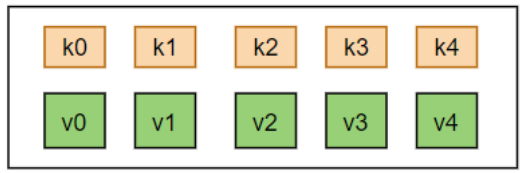
- 字典的定义和字典序列解包
- 字典的方法
- 函数定义：可变长参数
- 集合
- sorted函数的实践应用

字典字面量

- 列表、元组等序列对象按顺序存储多个对象，这些对象可以通过整数**下标**来快速访问
- 字典可存储多个对象，每个对象(value)通过一个唯一的键(key)标识并且快速访问
- 字典是包含键-值对(key-value pair)的映射(map)类型的可变序列： key映射到一个value
  - 字典中每个元素包含两部分：key和value。添加元素时必须同时指定key和其对应的value
  - 通过key可以很快地找到其映射的value: dictobj[key]
    - 不是按照key的大小顺序排列，具体顺序依赖于系统的内部实现(hashmap或hash table)
    - python3.7开始，记录插入顺序，遍历时按照插入的顺序
  - 已知对象(value)要找到其对应的key则需要许多额外的工作(遍历)
  - 键(keys)不允许（==关系意义上的）重复
    - 各个元素的key可以是不同类型，但必须是可hashable对象，即可调用hash(obj)得到一个值
    - 可变对象不能作为key
    - 整数、实数、复数、字符串等不可变对象可以作为key
    - 不可变的容器对象(如元组）有可能可以作为key，如(1,2)可以，但(1, 2, [1])不可以
  - 值(values)可是任何对象



s[0]
s[0] = 1
s.append(5)



d['one']
d['one'] = 1

创建字典：字典字面量(dict literal)

{key1:value1 ,key2:value2, … ,keyN:valueN}

- 大括号界定，元素用key:value表示，元素间用逗号分隔
- 按照先后顺序将元素加入字典，如果后面的元素的key与之前的一致，则更新其对应的值

```
>>> d = {}
>>> d1 = {'one':-1, 'two':2, 'three':3, 'one':1}
>>> d1
{'one': 1, 'two': 2, 'three': 3}
>>> d2 = {0:'food', 1:'drink', 2:'fruit'}
>>> d3 = {'name': 'Steve', 'age': 25, 'sex':'male',
         'address': {'city':'shanghai', 'zip':'200433'},
          1:'note1', 2:'note2',
          '1':'xx1',  '2':'xx2' }
```

```
>>> x = collections.OrderedDict()
>>> x['b'] = 5
>>> x['a'] = 3
>>> x['c'] = 8
>>> x
OrderedDict([('b', 5), ('a', 3), ('c', 8)])
```

遍历字典中的元素时：

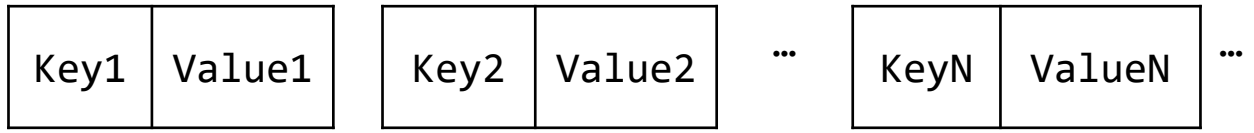
- Python3.6以前的版本不记录插入的顺序，字典的元素是按照key的hash值(即内部顺序)访问，所以打印输出与定义时的顺序会不一致,可能显示为： {'one': 1, 'three': 3, 'two': 2}
- Python3.6的实现记录插入的顺序，字典的元素按照**key插入到字典中的顺序**访问，显示为：
  {'one': 1, 'two': 2, 'three': 3}
- Python3.7： python语言明确字典记录插入的顺序
- 对于python3.7之前的版本，如果要求记录插入顺序，应该采用collections.OrderedDict

创建字典：构造函数dict(...)

dict()	创建空字典
dict(mapping)	从一个mapping对象(描述多个key:value对，比如字典) 创建一个新的字典
dict(iterable)	新字典的元素来自于iterable，iterable中的**每个元素必须是一个包含2个子元素的容器对象。**第一个子元素作为key，第二个子元素为value
dict(**kwargs)	根据函数调用时传递的**关键字参数(keyvar=value)**创建一个新的字典，该字典的key为字符串类型，该字符串保存了关键字参数传递给出的变量名，而对应的值为关键字参数传递的值，即对应的键值对为 **'keyvar':value**

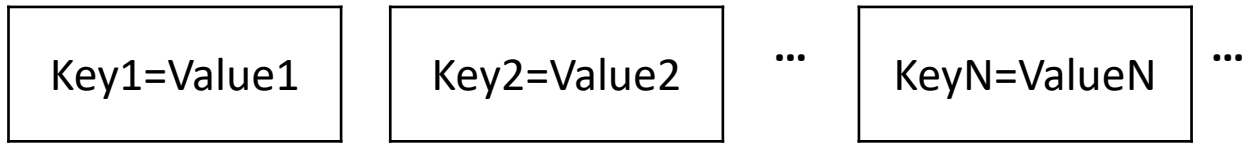
```
d3 = {'one': 1, 'two': 2, 'three': 3}
d4 = dict(d3)
d5 = dict([['one',1],('two',2),['three',3]])
z = zip(['one', 'two', 'three'], [1, 2, 3])
d6 = dict(z)
d7 = dict(one=1, two=2, three=3)
```

dict(iterable)



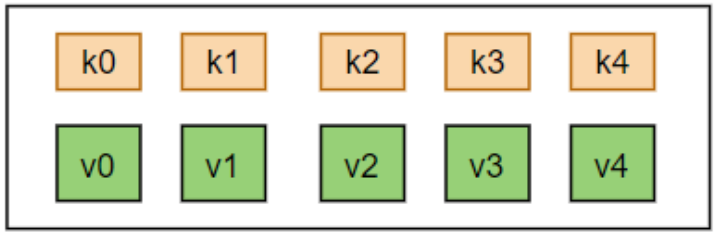
Key1:Value1　　Key2:Value2　　KeyN:ValueN

dict(Key1=Value1, Key2=Value2,...)



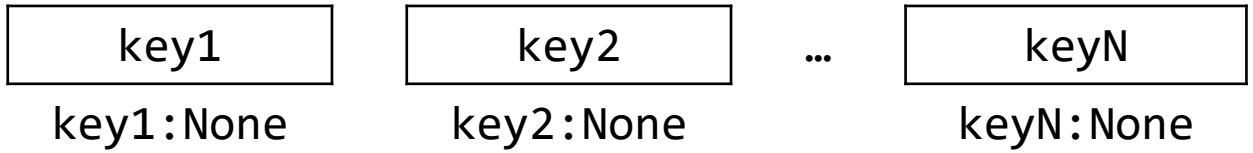
‘Key1’:Value1　　‘Key2’:Value2　　‘KeyN’:ValueN

创建字典：类方法fromkeys




```
dict.fromkeys(iterable[,value])
```

- 每个元素的key来自于可迭代对象，值设置为value，缺省为None



- 类方法
  - 通过类名调用方法，无需首先创建对象的实例
  - 当然也可创建对象实例，然后再调用类方法
- 实例方法：首先创建一个对象，然后通过该对象来调用方法

```
>>> d1 = dict.fromkeys(['name','age','sex'])
>>> d1
{'name': None, 'age': None, 'sex': None}
>>> d2 = dict.fromkeys(range(5), 10)
>>> d2
{0: 10, 1: 10, 2: 10, 3: 10, 4: 10}
>>> d3 = d1.fromkeys('123')
>>> d3
{'1': None, '2': None, '3': None}
```

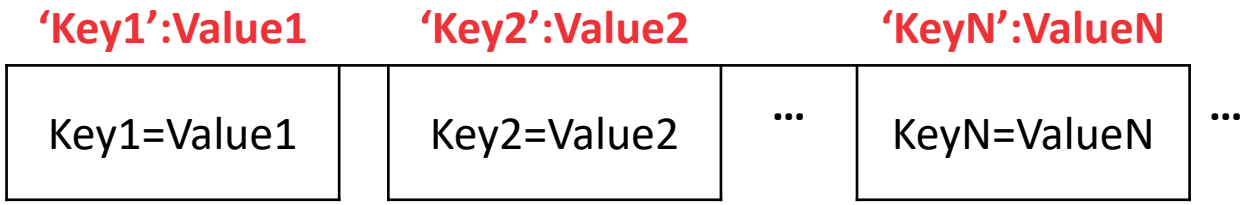
创建字典：字典解析式

- 列表解析式：[expr for value in iterable if condition]
- 生成表达式：(expr for value in iterable if condition)
- 字典解析式：{key:val for value in iterable if condition}

```
{k:v for k,v in zip(['a','b','c'],[1,2,3])}
# {'a':1, 'b':2, 'c': 3}
{c:c * 4 for c in 'abcd'}
# {'a':'aaaa','b':'bbbb','c':'cccc','d':'dddd'}
```

映射解包(mapping unpacking)

- 序列解包(iterable unpacking)可用在：
  - 赋值语句 x, y, z = 1, 2, 3; x, *y, z = range(10)
  - 函数调用: print(*[1, 2, 3])
  - 列表、元组字面量定义中： [*[1, 2, 3], 4, 5]
- 函数调用也支持映射解包
  - 函数调用时引入**dictobj。dictobj的**key必须为字符串类型**
  - 相当于传递多个关键字参数，每个关键字参数的名称为字典对象的**元素的key(字符串)包含的名字**(要满足变量名的要求），而关键字参数的值为字典对象中key对应的值
- 字典字面量定义时，引入**dictobj，表示将该字典展开成键值对的形式




```
d1 = dict(one=1, two=2, three=3)
d2 = dict(**d1)
d3 = dict(**{'one':1, 'two':2}, three=3)
d4 = {'one':1,  'two':2, 'three':3}
d5 = {**d4, 'four':4}
d6 = {'one':1,  'two':2, 'three':3, 'four':4}
```

```
d = {'k0':v0, 'k1':v1, 'k2':v2}
func(x, y, **d)
# 相当于func(x, y, k0=v0, k1=v1, k2=v2)
d = {k0:v0, k1:v1, k2:v2}
{'one':1, **d)
# 相当于 {'one':1,  k0:v0, k1:v1, k2:v2}
```

创建字典（总结）

字面量定义，在定义中也可采用字典的序列解包	{'name': 'Steve', 'age': 25, 'sex':'male'} {**{'one':1, 'two':2, 'three':3} , 'four':4}
• dict() • dict(iterable) • dict(**kwargs) • dict(mapping)	• 空字典 • 迭代对象中的元素必须是包括两个对象的容器对象，比如dict([('name','Steve'), ('age', 25), ('sex', 'male']) • 字典的key为字符串类型，包含关键字参数的名字，比如dict(name='Steve', age=25, sex='male') • 类似于原有mapping(即dict)的拷贝
dict.fromkeys(seq[,value])	seq中的元素作为字典中元素的key，而值为第二个参数或缺省的None，比如dict.fromkeys(('name', 'age', 'sex'), None)
{k:v for k,v in …}	{k:v for k,v in zip(('name', 'age', 'sex'), ('Steve', 25, 'male'))} 等价于 {'name': 'Steve', 'age': 25, 'sex': 'male'}

主要内容

- 字典的定义和字典序列解包
- **字典的方法**
- 函数定义：可变长参数
- 集合
- sorted函数的实践应用

字典支持的操作

已知key,是否有key:value对？对应的value是什么 | 遍历 | 插入或修改 | 删除

方法	说明
key in d **key not in d**	**判断字典d中有没有键==key，返回True or False** **相当于 key in d.keys()**
d[key]	返回key对应的value，key不存在raise KeyError
d.get(key[,default])	返回key对应的value，key不存在时返回default，缺省为None
d.keys()	返回可迭代对象，其中元素为字典中的所有key
d.values()	返回可迭代对象，其中元素为字典中的所有value
d.items()	返回可迭代对象，其中元素为包括了(key,value)的元组
d[key] = value	更新字典中键=key的元素的值为value，如果不存在，则添加key:value
setdefault(key[,default])	如果key不在字典，插入新元素，其值为default(缺省None)。如果在，不更新。返回d[key]
update(other)	根据另一字典或元素为key/value对的可迭代对象更新字典，返回None
del d[key]	**删除元素，如果key不存在，raise KeyError**
popitem()	**移走并返回某一个(key,value）对**
pop(key)	**如果key存在删除对应元素并返回值，否则raise KeyError**
pop(key,value)	**如果key存在删除对应元素并返回值，否则返回value**
clear()	**清除所有元素**
copy()	返回shallow copy后的新字典

字典方法：成员关系判断

key in dictobj和 key not in dictobj:

- 判断字典对象中有没有键==key，返回True or False

```
>>> d = {'name': 'Dong', 'sex': 'male', 'age': 37, 1: 'first', 2: 'second'}
>>> 'score' in d
False
>>> 'name' in d
True
```

- 字典支持 == 和!=运算，判断key:value对是否完全一致
- 不支持大小关系的比较，不支持 + - 等运算

```
>>> d1 = {0:0, 1:1}
>>> d2 = {1:1, 0:0}
>>> d1 == d2
True
>>> d1 > d2
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    d1 > d2
TypeError: '>' not supported between instances of 'dict' and 'dict'
```

字典方法：访问字典元素

dictobj[key]　　dictobj.get(key, default)

- d[key]：返回字典中key对应的value；key不存在时，抛出KeyError异常
- d.get(key[,default])： 返回字典中key对应的value；key不存在时返回default，缺省为None

```
>>> d = {'name': 'Tony', 'sex': 'male',
'age': 37, 1: 'first', 2: 'second'}
>>> d['name']
'Tony'
>>> d['score']
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    d['score']
KeyError: 'score'
>>> d[name]  # 表达式中包含变量
…
NameError: name 'name' is not defined
>>> d.get('sex', 'N/A')
'male'
>>> d.get('score', 'N/A')
'N/A'
>>> d.get('score')     # return None
```

```
if 'sex' in d:
    sex = '性别:' + d['sex'])
else:
    sex = '性别: 未知')
```

```
sex = '性别:' + d.get('sex', '未知')
```

字典方法：遍历字典

- 遍历时如果对应的字典对象的大小有改变时会抛出异常RuntimeError

len(d)	返回字典中元素的个数
d.keys()	返回**可迭代对象**，其中元素为字典中的所有key
d.values()	返回**可迭代对象**，其中元素为字典中的所有value
d.items()	返回**可迭代对象**，其中元素为包括了(key,value)的元组
iter(d)	返回迭代器对象，其中**元素为字典中的所有key**，类似于iter(d.keys()) 意味着d作为可迭代对象时，其元素为字典的key

```
d = {'name': ' 'Tony ', 'sex': 'male', 'age': 37}
```

```
for key in d.keys():
    print(key)
```

```
for key in d:
    print(key)
```

```
name
sex
age
```

```
for value in d.values():
    print(value)
```

```
Tony
male
37
```

```
for item in d.items():
    print(item)
```

```
('name', 'Tony')
('sex', 'male')
('age', 37)
```

```
for key,value in d.items():
    print(key,value)
```

```
name Tony
sex male
age 37
```

```
>>> d = {'name': 'Tony', 'sex': 'male', 'age': 37}
>>> d.items()
dict_items([('name', 'Tony'), ('sex', 'male'),
('age', 37)])
>>> list(d.items())
[('name', 'Tony'), ('sex', 'male'), ('age', 37)]
>>> list(d)
['name', 'sex', 'age']
```

字典方法：遍历字典示例

字典scores记录了同学的成绩，计算最高分、最低分、平均分，并查找所有最高分同学

```
>>> scores = {"Zhang San": 45, "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())
>>> lowest = min(scores.values())
>>> highest
99
>>> lowest
40
>>> average = sum(scores.values()) / len(scores)
>>> average
72.33333333333333
>>> highestPerson = [name for name, score in scores.items() if score == highest]
>>> highestPerson
['Wu Shi']
```

- for key in d: print(key, d[key])
- for k,v in d.items(): print(k, v)

[name for name in scores if scores[name] == highest]

[item[0] for item in scores.items() if item[1] == highest]

for item in scores.items():
    name, score = item

字典方法：修改字典

```
for key in another: # map对象
    dictobj[key] = another[key]
for key, value in another: # 可迭代对象
    dictobj[key] = value
```

d[key]=value

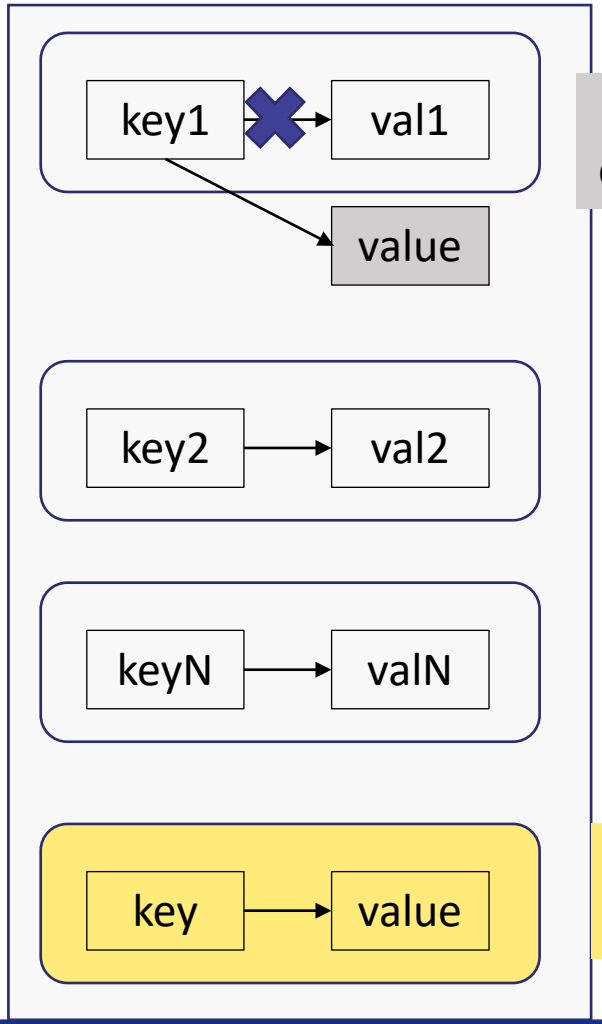
- 如果key存在，则**更新**字典中键=key的元素的值为value；如果key不存在，则**添加**key:value

dictobj.update(another)

dictobj.update(key1=value1,key2=value2)

- update的参数与dict()的参数类似
  - 字典another: 将another中的key:value对加入到当前字典对象
  - 可迭代对象another: 要求其元素为包含两个元素的容器对象
  - 可采用关键字参数传递： key=value

```
>>> d = {'name':'Tony', 'sex':'male', 'age':37}
>>> d['age'] = 38  #  {'name': 'Tony', 'sex': 'male', 'age': 38}
>>> d['city'] = 'BeiJing'
>>> d.update({'age':39, 'city':'Shanghai'})
>>> d
{'name': 'Tony', 'sex': 'male', 'age': 39, 'city': 'Shanghai'}
>>> d.update([('age', 40),('phone','1234')])
>>> d.update(age=41)   # 采用关键字参数传输
>>> d
{'name': 'Tony', 'sex': 'male', 'age': 41, 'city': 'Shanghai',
'phone': '1234'}
```



更新key:value
d[key1] = value

添加key:value
d[key] = value

字典方法：访问元素与修改元素结合

get(key[, default])　　返回d[key]或者default

setdefault(key[,default])　返回d[key]或者default，但是如果key不在字典，会首先添加key:default

- 如果key在字典里，则返回d[key]
- 如果**key不在字典**里插入新元素，其值为default(缺省None)，然后返回d[key]
- 相比get(key[,default])而言，如果key不在字典里时会首先初始化key对应的value为default

```
# get(key[, default])
if key in d:
    return d[key]
else:
    return default
```

```
# setdefault(key[, default])
if key in d:
    return d[key]
else:
    d[key] = default
    return d[key]
```

字典应用示例：value为不可变对象

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。

- key为某个字符
- value为该字符出现的次数

letter_counter.py

```
def letter_counter(seq):
    counters = dict()   # counters = {}
    for item in seq:
        counters[item] = counters.get(item, 0) + 1
    return counters
```

```
def test_letter_counter():
    import string
    import random
    x = string.ascii_letters + string.digits + string.punctuation
    y = [random.choice(x) for i in range(1000)]
    z = ''.join(y)      # 将可迭代对象y中的元素合并成一个字符串
    print('测试字符串…')
    print(z)
    counters = letter_counter(z)
    print('统计结果为:',counters)
    for letter in counters:    #字典作为可迭代对象，元素为key
        print('%s的出现次数==>%d' % (letter, counters[letter]))
```

```
counters[item] = counters.setdefault(item, 0) + 1
# 多赋值一次，如果item不存在，首先item:0，然后item:1
```

```
if item in counters:
    counters[item] += 1
else:
    counters[item] = 1
```

字符串的join方法：组合多个字符串

strobj.join(iterable)：

- 将iterable对象(列表,字符串等)中的多个元素组合在一起，元素间插入相应的**分割字符串strobj**，返回新的字符串
- iterable对象中的**元素必须是字符串**

```
>>> s = list('abc')
>>> s
['a', 'b', 'c']
>>> ''.join(s)
'abc'
>>> ','.join(s)
'a,b,c'
>>> print('\n'.join(s))
a
b
c
>>> list2 = list(range(5))
>>> list2
[0, 1, 2, 3, 4]
>>> '+'.join([str(i) for i in list2])
'0+1+2+3+4'
```

字典应用示例：value为可变对象

字典对象d记录了多次考试的分数。key为某个同学的姓名，而对应的value为一个列表

假设某次考试成绩出来了，要保存名为name的同学的成绩score

```
name, score = 'tony', 98
```

- 如果以前有过成绩(key in d)，则首先获得原有成绩列表，将最新成绩原地附加到该列表
- 如果没有成绩，则创建一个新列表，添加成绩，然后加入到字典中

```
# setdefault(key[, default])
if key in d:
    return d[key]
else:
    d[key] = default
    return default
```

dict_update_mutable.py

```
d = { }

if name in d:
    d[name].append(score)
else:
    d[name] = [score]

d[name] = d.get(name, [])
d[name].append(score)

d = { }
d[name] = d.get(name, []) + [score]

d.setdefault(name,[]).append(score)
```

```
# 下述代码有错！ append返回None，等价于
d1_err[name] = None
```

```
d1_err = { }

d1_err[name] = d1_err.get(name, []).append(score)
```

使用加法运算的版本性能最差，每次都创建新的列表

package collections

collections引入了更多的类型来支持更多的数据结构

defaultdict	特殊的字典，在没有指定value时采用缺省值factory()
OrderedDict	有序的字典，保留元素插入的顺序
Counter	保存对象出现次数的字典
ChainMap	用于将多个字典组合在一起，提供类似于字典的接口
UserDict	封装了一个字典对象，便于扩充编写自己定制的字典
deque	双向队列，提供类似于列表的接口，在队列头和尾部增加和删除元素时非常快
UserList	封装了一个列表对象，便于扩充编写自己定制的列表
UserString	封装了一个字符串对象，便于扩充编写自己定制的字符串
namedtuple	特殊的元组，可以通过名字来访问元组中的元素

defaultdict

模块collections中引入了一个特别的字典类型defaultdict

- defaultdict(factory[,...])用于创建一个defaultdict对象
  - 第一个参数为一个**可调用对象factory**（函数），其余参数与dict函数的参数一样
  - 字典的key:value对的value缺省值为 **调用factory()所返回的结果**
- dict的所有方法都可用于defaultdict
- 在通过dictobj[key]访问元素时：
  - 如果key存在，返回dictobj[key]
  - 如果key不存在，会首先设置dictobj[key]为一个缺省值，然后返回dictobj[key]

    dictobj[key] = value = factory(); return value
  - 副作用是如果不小心写错了key，会增加一个新的元素

```
# defaultdict:  d[key]
if key in d:
    return d[key]
else:
    d[key] = factory()
    return d[key]
```

字典对象d记录了多次考试的分数。key为某个同学的姓名，而对应的value为一个列表
假设某次考试成绩出来了，要保存名为name的同学的成绩score

```
name, score = 'tony', 98
```

dict_update_mutable.py

```
from collections import defaultdict
d5 = defaultdict(list)  #创建一个空字典，值的缺省值为 list()，即空列表
d5['tony'].append(98)
```

defaultdict：字典应用示例：value为不可变对象

- 使用collections模块的defaultdict类（提供缺省值的dict）

letter_counter.py

```
def letter_counter(seq):
    counters = dict()
    for item in seq:
        counters[item] = counters.get(item, 0) + 1
    return counters
```

```
def letter_counter2(seq):
    from collections import defaultdict
    counters = defaultdict(int)
    for item in seq:
        counters[item] += 1
    return counters
```

缺省值为 int()，即为0

Counter

- collections中的Counter是一种特殊的字典，在传递非map类型的可迭代对象时，有特别的处理，保存了各个元素出现的次数
- 在调用构造函数创建Counter对象时:
  - 如果传递的为**非map类型的可迭代对象**，则创建的Counter对象中，key为可迭代对象的元素，而**value为该key在原可迭代对象中出现的次数 即iterable.count(key)**
  - 如果是其他类型的参数，则与字典构造函数类似：
    - 如果传递的为map类型，则相当于原有map类型的shallow copy
    - 如果通过关键字参数传递，与原有dict类型类似，等同于构造一个字典

e1	e2	e3	e3	e2	e4	e1	e5

Counter(seq)创建的Counter对象对应： {e1:2, e2:2, e3:2, e4:1, e5:1}

letter_counter.py

```
def letter_counter(seq):
    counters = dict()
    for item in seq:
        counters[item] = counters.get(item, 0) + 1
    return counters
```

```
def letter_counter(seq):
    from collections import Counter
    return Counter(seq)
```

Counter

拓展的内容

- 特殊的字典，原有字典中的方法(get,setdefault, keys,values,items,clear, pop,popitem)都可以使用
- 扩展了update(another)方法：不是替代原有的key:value，而是在原有的value基础上加上another中相应元素出现的次数
- 引入subtract(another)：当前计数器对象中的元素出现的次数减去another对象中元素出现的次数
- 支持与另外一个Counter对象的加减法运算，包括+= -= 以及 + - 运算符
- elements()：返回一个迭代器，其元素为Counter中的key，如果该key对应的值大于1，则key会重复出现相应的次数
- most_common(n=None) : 返回一个n个元素的列表，每个元素为(key, value)，按照key出现的次数(value) 从大到小排列，即**返回前面n个出现次数最多的(key,value)**。如果n=None，返回包括所有元素的列表

```
>>> from collections import Counter
>>> c = Counter('abcdeabcdabcaba')
>>> c
Counter({'a': 5, 'b': 4, 'c': 3, 'd': 2, 'e': 1})
>>> c.elements()
<itertools.chain object at 0x00000224DFEA1048>
>>> list(c.elements())
['a', 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b', 'c', 'c',
'c', 'd', 'd', 'e']
>>> c.most_common(3)
[('a', 5), ('b', 4), ('c', 3)]
>>> c['f']    # 'f'不存在时不会抛异常
0
>>> c
Counter({'a': 5, 'b': 4, 'c': 3, 'd': 2, 'e': 1})
>>> c.update('befabcbbccd')
>>> c
Counter({'b': 8, 'a': 6, 'c': 6, 'd': 3, 'e': 2, 'f': 1})
```

字典方法：读取和修改

d[key]	返回key对应的value，key不存在raise KeyError
d.get(key[,default])	返回key对应的value，key不存在时返回default，缺省为None
d[key] = value	更新字典中键=key的元素的值为value，如果不存在，则添加key:value
setdefault(key[,default])	如果key不在字典里插入新元素，其值为default(缺省None)。如果在不更新。返回d[key]
update(other)	根据另一字典或元素为key/value对的可迭代对象更新字典，返回None
d=defaultdict(list) **d[key]**	key如果不存在时，会自动添加key:list()到字典中，即 d[key] = list()

字典方法：删除元素

del d[key]	删除元素，如果key不存在，raise KeyError
pop(key)	如果key存在，删除对应元素并返回值，否则raise KeyError
pop(key,value)	如果key存在，删除对应元素并返回值，否则返回value
popitem()	移走并返回某一个(key,value）对，如果为空 raise KeyError
clear()	清除所有元素

```
>>> d ={ 'age': 38, 'score': [98, 97], 'name': 'Tony', 'city': 'shanghai', 'sex': 'male'}
>>> del d['city']
>>> d
{'age': 38, 'sex': 'male', 'name': 'Tony', 'score': [98, 97]}
>>> d.pop('sex')
'male'
>>> d.pop('sex')
Traceback (most recent call last):
 … d.pop('sex')
KeyError: 'sex'
>>> d.pop('score', [])
[98, 97]
```

```
>>> d.pop('score', [])
[]
>>> d
{'age': 38, 'name': 'Tony'}
>>> d.popitem()
('name', 'Tony')
>>> d.clear()
>>> d.popitem()
…
   d.popitem()
KeyError: 'popitem(): dictionary is empty'
```

字典方法：删除元素示例

函数parse_options支持关键字实参传递verbose/reverse/key，如果没有传递时相当于：verbose=False, reverse=False, key=None

parse_kwargs.py

```
def parse_options(**kwargs):
    print(kwargs)
    verbose = kwargs.pop('verbose', False)
    reverse = kwargs.pop('reverse', False)
    key = kwargs.pop('key', None)
    while len(kwargs):
        option, value = kwargs.popitem()
        print('unknown option:%s=%s' % (option, value))

if __name__ == '__main__':
    options = {'reverse':True, 'key':lambda item:item[0], 'option1':1, 'option2':2}
    parse_options(**options)
```

```
parse_options(reverse=True, key=lambda item:item[0], option1=1, option2=2)
```

- parse_options(**options) 函数调用时的序列解包，展开为关键字参数
- def parse_options(**kwargs) 函数定义时的可变长度字典参数，收集关键字参数，组成一个字典赋值给kwargs

主要内容

- 字典的定义和字典序列解包
- 字典的方法
- **函数定义：可变长参数**
- 集合
- sorted函数的实践应用

函数定义：可变长参数

- 函数定义时，参数（形参）可以是：
  - 普通(位置)参数
  - 缺省值参数，相应位置没有参数传递时使用缺省值
  - **可变长度位置参数 *args，调用时相应位置可以传递0个或者多个位置参数（实参）**
  - **可变长度关键字参数 **kwargs，调用时可以传递0个或多个关键字参数（实参）**
  - **keyword-only参数，如果要传递值时只能通过关键字参数（实参）方式传递**
- 函数调用时，参数（实参）可以是：
  - 普通（位置）参数
  - 关键字参数

```
def var_func (a, b=4, *x, **y):
    return (a, b, x, y)

print(var_func(1,2,3,4,5,u=11,v=12,w=13))
```

- Python函数的定义非常灵活，在定义函数时只需要指定参数的名字，而不需要指定参数的类型
  - 传递某些参数时可能执行正确，而传递另一些类型的参数时可能出现错误
  - 函数编写如果有问题，只有在调用时才能被发现
- **type hint**: 给变量、参数和返回值加上annotations，集成开发环境可检查调用和执行过程中是否有类型错误

```
def func(text: str, times: int) -> None:
    print(text * times)
    return None
```

函数定义：可变长参数

可变长度参数在定义函数时主要有两种形式：

*parameter：将那些尚没有匹配形参的**位置实参**组合成一个**元组**赋值给可变长度位置参数

parameter：将那些尚没有匹配形参的关键字实参**组合成一个**字典**赋值给可变长度关键字参数。字典的key为字符串，其内容为关键字实参名，而值为传递的实参对象

- 如果存在**可变长度位置参数**，*name形式的参数应该**出现在缺省值参数之后**
- 如果**存在可变长度关键字参数**，**name形式的参数**应该出现在最后**，即在位置参数、缺省值参数(如果有)、可变长度位置参数(如果有)之后
- 每种类型的可变长度参数只能出现一次
- 函数定义时：位置参数→缺省值参数→可变长度位置参数→可变长度关键字参数

variable_parameter.py

```
def variable_parameter(a, b=4, *x, **y):
    return (a, b, x, y)

def test_variable_parameter():
    print(variable_parameter(1))
    print(variable_parameter(1,2))
    print(variable_parameter(1,2,3,4,5))
    print(variable_parameter(1,u=11,v=12,w=13))
    print(variable_parameter(1,2,3,4,5,u=11,v=12,w=13))
```

```
(1, 4, (), {})
(1, 2, (), {})
(1, 2, (3, 4, 5), {})
(1, 4, (), {'u': 11, 'v': 12, 'w': 13})
(1, 2, (3, 4, 5), {'u': 11, 'v': 12, 'w': 13})
```

函数定义：仅允许关键字传递的形参（keyword-only parameters）

Python3还引入了**keyword-only形参**，**在调用时只能通过关键字实参**的形式来传递

- keyword-only参数**出现在可变长度的位置参数之后，可变长度关键字参数(如果有）之前**
  - 如果没有可变长度位置参数，则添加一个*
- keyword-only参数也可以指定缺省值，即支持name或name=default方式
- 调用时必须采取关键字参数传递，或者不传递时表示采用缺省值(如果定义时有缺省值)

kwonly.py

```
def kwonly(a, *b, c, d=5):
    return (a,b,c,d)

def kwonly2(a, b, *, c, d=5):
    return (a,b,c,d)

print(kwonly(1, 2, c=3))
print(kwonly2(1, 2, c=3, d=4))
```

```
m = max(4, 5, 6)
n = min(4, 5, 6)
print(m, n, sep=', ', end='\n')
```

keyword-only

函数定义：形参与实参之间的匹配

- 函数定义中的形参顺序

位置参数 →缺省值参数→可变长度位置参数→仅允许关键字传递参数(可定义缺省值，也可不定义)→可变长度关键字参数

- 函数调用中的实参顺序：位置参数→关键字参数
- 函数调用时，根据传递的实参按照以下规则匹配形参：
  - 首先按照位置匹配形参中的**位置参数和缺省值参数**
  - 接下来根据关键字参数的名字匹配形参中的各个参数
  - 剩下的位置参数组成tuple赋值给可变长度位置参数
  - 剩下的关键字参数组成dict赋值给可变长度关键字参数
  - 函数定义中尚未匹配的参数设置为缺省值
  - 如果仍然有尚未匹配的形参和实参则报错
  - 每个参数只能匹配一次

```
def func1(p1,p2,d1='v1',d2='v2',*var,
  k1,k2='kv2',**mapping):
    print(p1,p2,d1,d2,var,k1,k2,mapping)

func1(*(1,2,3,4,5,6,7),k1='hello k1',k3=4,k5=7)
```

```
# 1 2 3 4 (5, 6, 7) hello k1 kv2 {'k3': 4, 'k5': 7}
```

函数定义：可变长参数示例

编写函数，接收任意多个实数，返回一个**元组**，其中第一个元素为所有参数的平均值，其他元素为所有参数中大于平均值的实数

large_reals.py

```
def demo(*para):
    avg = sum(para) / len(para)
    g = [i for i in para if i > avg]
    return (avg,) + tuple(g)

print(demo(1, 2, 3, 4))
```

也可采用元组定义时的序列解包

```
return (avg, *g)
```

函数定义：可变长参数和keyword-only参数示例

- **自己定义 _print函数，该函数实现了内置函数print类似的功能**

```
def _print(*args, sep=' ', end='\n', file=sys.stdout, flush=False):
    line = []
    for value in args:
        line.append(format(value))
    file.write(sep.join(line))
    file.write(end)
    if flush:
        file.flush()
```

_print.py

内置函数format(value, format_spec='') 将value进行格式化，缺省转换为字符串

注意函数定义时的可变长参数与序列解包的区别

- 函数定义时的可变长参数：收集尚未匹配的位置参数或关键字参数
- 函数调用时的序列解包：展开为位置参数或关键字参数

函数定义：可变长参数与序列解包示例

_trace函数: 跟踪用户自定义的函数的调用，利用函数对象的属性(这里使用calls)来保存函数调用的状态信息(调用了多少次)。注意内置函数不支持用户自定义的属性

```
def _trace(func,*args,**kwargs):
    if hasattr(func,'calls'): #对象func是否有属性calls存在
        func.calls += 1
    else:
        func.calls = 1
    if kwargs:
        print('[%d] calling function: %s(%s,%s)'% (func.calls, func.__name__,
  ','.join([str(item) for item in args]),  ','.join([k+'='+str(v) for k,v in
kwargs.items()])))
    else:
        print('[%d] calling function: %s(%s)'% (func.calls, func.__name__,
                  ','.join([str(item) for item in args])))
    return func(*args,**kwargs)
```

- hasattr(obj, name) 检查obj是否有名字为name的属性存在

_trace.py

```
def demo(a,b=4,*x, **y):
    print(a, b, x, y)

if __name__ == '__main__':
    _trace(demo, 1)
    _trace(demo, 1, 2)
    _trace(demo, 1, 2, 3, 4, c=9, d=8)
```

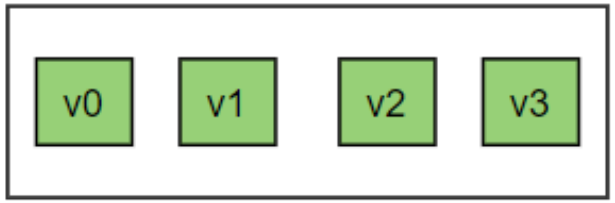
```
[1] calling function: demo(1)
1 4 () {}
[2] calling function: demo(1,2)
1 2 () {}
[3] calling function: demo(1,2,3,4,c=9,d=8)
1 2 (3, 4) {'c': 9, 'd': 8}
```

主要内容

- 字典的定义和字典序列解包
- 字典的方法
- 函数定义：可变长参数
- **集合**
- sorted函数的实践应用

集合

- 集合是无序**可变对象**，也是一个可迭代对象
  - 集合中的元素不重复，且无序
  - 集合中的元素必须是可hash对象，不能是可变对象
- 集合字面量：
  - 使用一对大括号界定，元素间以逗号分隔
  - 定义时可以包含重复的对象，但**只会保留一个**
- 构造函数：set([iterable])
  - 创建一个set对象，其元素为iterable对象中的元素，同样重复的元素仅保留一个
- {}定义的是空字典，而不是空集合，**set()**创建空集合对象




```
>>> {3, 7, 5, 5}
{3, 5, 7}
>>> {3, [1, 2]}
...
TypeError: unhashable type: 'list'
>>> set(range(8, 14))
{8, 9, 10, 11, 12, 13}
>>> set([0, 1, 2, 3, 0, 1, 2, 3, 7, 8])
{0, 1, 2, 3, 7, 8}
>>> set()
set()
```

duplicated_items.py

```
import random
s = [random.randint(0,100) for i in range(100)]
print(s)
if len(set(s)) == len(s):
    print('no duplicated scores')
else:
    print('duplicated scores!')
```

集合的运算

整数的位运算符:逐个二进制比特运算
- 或运算 | 全0才为0，**否则为1**
- 与运算 & **全1才为1**，否则为0
- 异或运算 ^ 半加，相等为0，否则1

- 表格中的运算符或者方法不是原地修改，而是返回一个新的集合对象

方法	说明	例子
s1 | s2 | … s1.union(s2,…)	并集，s1 ∪ s2 ∪ … 在任一一个集合	{1,2,3} | {2,4} , return { 1,2,3,4}
s1 & s2 & … s1.intersection(s2,…)	交集，s1 ∩ s2 ∩… 在所有集合出现	{1,2,3} & {2,4} , return {2}
s1 – s2 s1.difference(s2)	差集，s1 – s2 在s1但不在 s2	{1,2,3} - {2,4} return { 1,3}
s1 ^ s2 s1.symmetric_difference(s2)	对称差集, s1 ⊕ s2 只属于其中一个但不属于其中另一个 = (s1 ∪ s2) - (s1 ∩ s2)	{1,2,3} ^ {2,4} return { 1, 3, 4}
s1.isdisjoint(s2)	没有共同元素为True	{1,2,3} .isdisjoint({2,4}) return False
s1.issubset(s2) s1 <= s2	s1是否为s2的子集	{1,2} .issubset({1,2,4}) return True
s1.issuperset(s2) s1>= s2	s1是否为s2的超集	

集合：比较运算符

- 比较：
  - == != 判断两个集合是否相等
  - < > 判断前一集合是否是后者的真子集和真超集
  - <= >= 判断前一集合是否是后者的子集和超集
- 元素是否存在：
  - x in s 或者 x not in s: 判断元素x是否在集合s中
- len(s)：集合的元素个数

集合的方法：原地修改

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
```

方法	说明	例子
s.add(x)	将x添加到集合s	s1.add('a') # s1 = {1,2,3,'a'}
s1.update(s2)	并集 s1 =s1 ∪ s2	s1.update(s2) # s1 = {1,2,3,4}
s1.intersection_update(s2)	交集 s1 = s1 ∩ s2	s1.intersection_update(s2) # s1 = {2,3}
s1.difference_update(s2)	差集 s1 = s1 – s2	s1.difference_update(s2) # s1 = {1}
s1.symmetric_difference_update(s2)	对称差集 s1 = s1 ⊕ s2	s1.symmetric_difference_update(s2) # s1 = {1,4}
s.remove(x)	从集合s中移除x，不存在抛异常KeyError	s1.remove(1) # s1 = {2,3}
s.discard(x)	从集合s中移除x	s1.discard(1) # s1 = {2,3}
s.pop()	从集合s移除一个元素并返回，空时KeyError	s1.pop() # return 1, s1 = {2,3}
s.clear()	清空集合s	s1.clear() # s1 = set()

集合解析式和frozenset

- 类似于列表解析式：{expr for item in iterable if expr}

```
>>> {2*x for x in range(5)}
{0, 8, 2, 4, 6}
```

不可变集合frozenset

- set是可变对象
- frozenset为**不可变**集合对象，只能通过forzenset()函数创建，与set()函数类似
- 除了集合元素的更改类的操作不支持外，set的其他操作都可应用于frozenset

集合使用示例：去除重复元素

```
>>> s = [0, 11, 8, 7, 9, 10, 2, 3, 3, 13]
>>> list(set(s))
[0, 2, 3, 7, 8, 9, 10, 11, 13]
```

unique_list.py

```
import random
# 产生100个[0,10000)之间的随机整数，保存在列表中
listRandom = [random.choice(range(10000)) for i in range(100)]
```

```
# 循环+选择结构
noRepeat0 = []
for i in listRandom :
    if i not in noRepeat0 :
        noRepeat0.append(i)
```

- 利用**set不重复特性去除重复元素**，但是无序特性导致产生的list不是按照原来的顺序
- **集合实现并不保留插入的顺序**，而是一个内部的顺序

```
noRepeat = list(set(listRandom))
```

```
newDict = dict.fromkeys(listRandom)
noRepeat2 = list(newDict)
print(*noRepeat2)

import collections
newDict = collections.OrderedDict.fromkeys(listRandom)
noRepeat3 = list(newDict)
print(*noRepeat3)
```

利用**dict的key的不重复特性去除重复元素**

- Python3.6以后的dict实现是按照key插入的顺序
- Python3.6之前使用collections.OrderedDict保留原来的顺序

集合使用示例：集合的差集

parse_kwargs.py

```
def parse_options(**kwargs):
    print(kwargs)
    verbose = kwargs.pop('verbose', False)
    reverse = kwargs.pop('reverse', False)
    key = kwargs.pop('key', None)
    while len(kwargs):
        option, value = kwargs.popitem()
        print('unknown option:%s=%s' % (option, value))

def check_options(**kwargs):
    valid_options = {'verbose', 'reverse', 'key'}
    unknown_options = set(kwargs) - valid_options
    if len(unknown_options):
        print('unknown options:', *unknown_options)

if __name__ == '__main__':
    options = {'reverse':True, 'key':lambda item:item[0], 'option1':1, 'option2':2}
    check_options(**options)
    parse_options(**options)
```

parse_options(verbose=False, reverse=False, key=None)

集合A - 集合B = {在集合A但不在集合B中的元素}

check_options(reverse=True,key=lambda ite:item[0], option1=1, option2=2

主要内容

- 字典的定义和字典序列解包
- 字典的方法
- 函数定义：可变长参数
- 集合
- **sorted函数的实践应用**

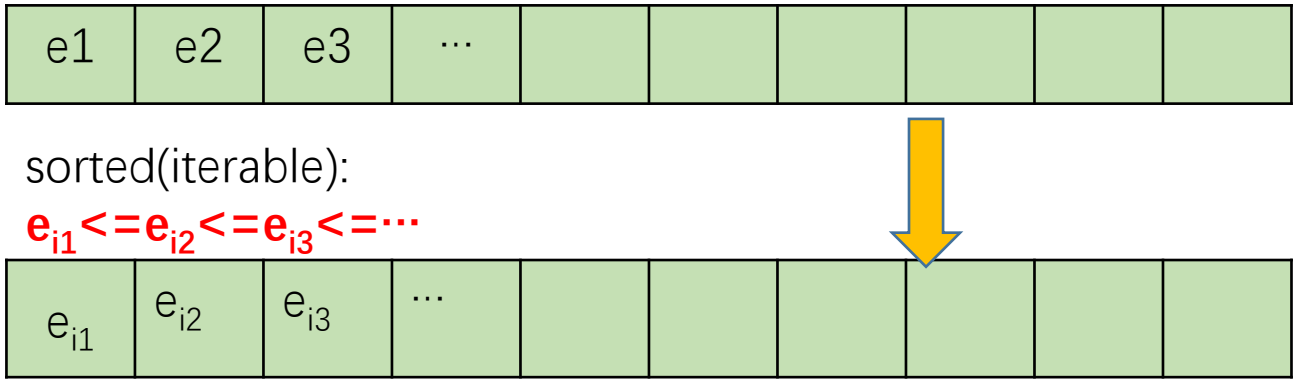
内置函数sorted的实践应用

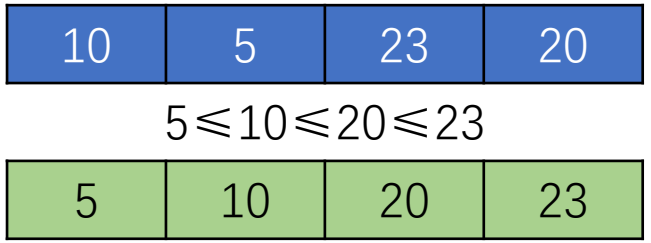
L.sort(key=None, reverse=False)　原地排序列表，返回**None**

sorted(iterable, key=None, reverse=False) 对可迭代对象的元素排序，返回新的列表

- key缺省为None，表示基于元素间的大小关系排序
- key也可传递函数对象，作为排序的基准
  - 在调用时传递一个参数，该参数为要排序的iterable对象中的某个元素
  - 函数调用的返回值作为给iterable对象中对应元素排序的基准
  - 排序算法是稳定的，在基准相同时按照原有出现的顺序排列

- 新列表中的元素来自于可迭代对象
- 排序的基准是调用关键字参数所传递的函数key(item)得到的结果






```
s = [10, 5, 23, 20]
s1 = sorted(s)
print(s1)
```

内置函数sorted的实践应用

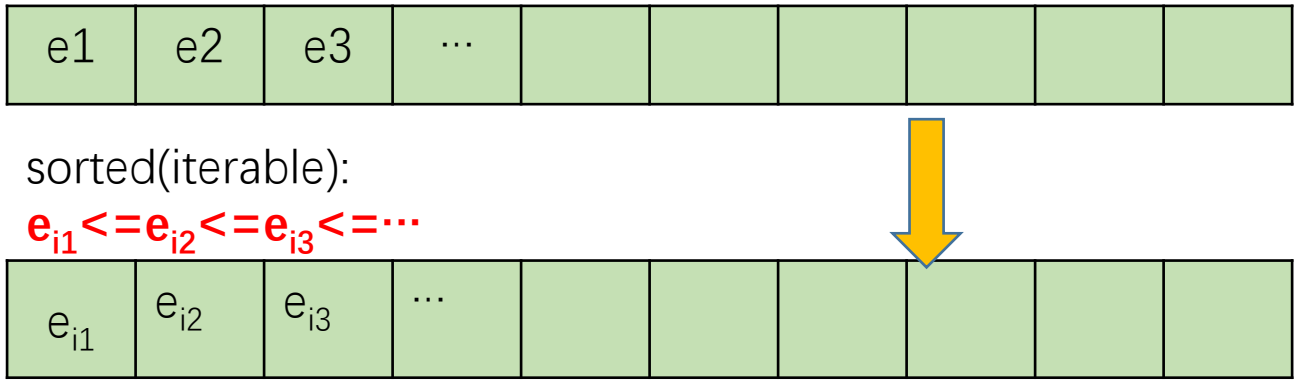
sorted(iterable, key=None, reverse=False)

key缺省为None，表示基于元素间的大小关系排序

key也可传递函数对象，作为排序的基准

- 排序算法是稳定的，在基准相同时按照原有出现的顺序排列

```
[4, 5, 3, 6, 13, 2, 8, 9, 10, 11, 0, 1, 7, 12]
[4, 5, 3, 6, 2, 8, 9, 0, 1, 7, 13, 10, 11, 12]
[4, 5, 3, 6, 2, 8, 9, 0, 1, 7, 13, 10, 11, 12]
```



sorted(iterable, key=key):

$key(e_{i1})<=key(e_{i1})<=key(e_{i1})<=\cdots$

```
s = [10, 5, 23, 20]
def weight(item):
    item = str(item)
    return int(item[::-1])
s2 = sorted(s, key=weight)
print(s2) # [10, 20, 5, 23]
```

排序基准：原有整数的数字逆序后得到的整数

sorted_demo.py

```
import random
random.seed(1234)
s = list(range(14))
random.shuffle(s)
print(s)
ss = s.copy()

def weight(item):
    return len(str(item))

s.sort(key=weight)
print(s)

ss.sort(key=lambda item: len(str(item)))
print(ss)
```

排序基准： 整数的位数

内置函数sorted的实践应用

```
sorted(iterable, key=None, reverse=False)
```

排序要解决两个问题：

- 对什么进行排序?
  - 列表的sort方法对于自身进行排序
  - 内置函数sorted返回新的列表，传递的可迭代对象决定了新列表中的元素
    - 传递的参数为元组、列表、集合，则新列表中的元素就是这些对象中原来的元素
    - 传递的参数为字典，则新列表中的元素就是字典的key
      - 字典对象通过key可以得到其对应的值，sorted传递的key参数可以利用这一点
    - 传递的参数为dictobj.items()，排序后的元素为元组(key,value)
- 怎么排序?
  - 按照元素之间的大小顺序比较
  - 定义key参数,根据key(item)返回的值来确定顺序
    - 可以使用def语句定义一个新的函数，然后传递该函数对象
    - 可以使用lambda表达式定义一个匿名函数对象

内置函数sorted的实践应用

对于persons进行排序返回一个新的列表：先按姓名升序排序，姓名相同的按年龄降序排序

```
>>> persons = [{'name':'Dong', 'age':37}, {'name':'Zhang', 'age':40}, {'name':'Li', 'age':50}, {'name':'Dong', 'age':43}]
# 期望的列表： [{'name': 'Dong', 'age': 43}, {'name': 'Dong', 'age': 37}, {'name': 'Li', 'age': 50}, {'name': 'Zhang', 'age': 40}]
```

1 使用def语句定义一个函数对象，然后使用key来指定排序依据

- 定义一个函数对象sort_by_name_age，该函数调用时传递的参数为要排序的元素（这里是字典对象），返回两元元组，第一个元素为姓名，第二个元素为年龄的负值
- 元组的比较：第一个元素比较大小，如果相同的情况下比较第二个元素，如此继续…

```
def sort_by_name_age(item):
    return item['name'],-item['age']
print(sorted(persons, key=sort_by_name_age))
```

内置函数sorted的实践应用

- 对于persons进行排序返回一个新的列表：先按姓名升序排序，姓名相同的按年龄降序排序

```
>>> persons = [{'name':'Dong', 'age':37}, {'name':'Zhang', 'age':40}, {'name':'Li', 'age':50}, {'name':'Dong', 'age':43}]
# 期望的列表： [{'name': 'Dong', 'age': 43}, {'name': 'Dong', 'age': 37}, {'name': 'Li', 'age': 50}, {'name': 'Zhang', 'age': 40}]
```

2 采用lambda表达式定义一个匿名函数对象，该函数参数为item，而冒号后面的为要返回的表达式，即(item['name'], -item['age'])

```
def __anonymous_function__(arguments):
    return expr
```

```
lambda arguments: expr
```

```
>>> print(sorted(persons, key=lambda item:(item['name'], -item['age'])))
```

注意上述sorted函数调用如果在item:后面少一对括号，即为：

sorted(persons, key=lambda item:item['name'], -item['age'])

第一个参数为persons

第二个参数为key=lambda item:item['name']

第三个参数为 -item['age']

会报语法错： SyntaxError: positional argument follows keyword argument

lambda表达式的优先级最低，大部分情况下expr不用加上括号，但注意如果表达式中有**省略了圆括号的元组定义时**，容易出现问题

内置函数sorted的实践应用

```
>>> scores = [['Bob', 95.0, 'A'], ['Alan', 86.0, 'C'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
```

采用列表，列表中的元素是一个列表（容器对象），包含姓名、分数和等级

- 排序什么？对原有的列表中的元素进行排序
- 怎么排序？元素本身的大小关系，或者key参数（定义函数、采用lambda表达式）

 # 按姓名升序，姓名相同按分数升序排序

```
>>> sorted(scores)
[['Alan', 86.0, 'C'], ['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
>>> sorted(scores, key=lambda item:(item[0], item[1]))
[['Alan', 86.0, 'C'], ['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
```

 # 按分数升序，分数相同的按姓名升序排序

```
>>> sorted(scores, key=lambda item:(item[1], item[0]))
[['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E'], ['Bob', 95.0, 'A']]
```

内置函数sorted的实践应用

```
>>> phonebook = {'Linda':'7750', 'Bob':'9345', 'Carol':'5834'}
```

如果希望基于电话号码进行排序，得到每个人的姓名和电话号码

1. sorted的第一个参数可以是phonebook.items()

```
>>> sorted(phonebook.items(), key=lambda item:item[1])
>>> def sort_by_item_1(item): return item[1]
>>> sorted(phonebook.items(), key=sort_by_item_1)
```

- 传递的参数为字典，则新列表中的元素就是字典的key
  - 字典对象通过key可以得到其对应的值，sorted传递的key参数可以利用这一点
- 传递的参数为dictobj.items()，排序后的元素为元组(key,value)

2. sorted的第一个参数可以是phonebook

```
>>> names = sorted(phonebook, key=lambda item:phonebook[item])
>>> [(name, phonebook[name]) for name in names]
```

```
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
```

内置函数sorted的实践应用

根据另外一个列表list2的值来对当前列表list1元素进行排序

- 假设cities保存的是城市的名称，而temperatures对应位置的元素为该城市的最高温度
- 现在希望按照温度的顺序返回城市的名称的列表(即根据temperatures来对cities排序)

Shanghai	Beijing	Wuhan	Chengdu	Nanjing

30	26	34	29	38

zip(temperatures,cities)

30,	Shanghai
26,	Beijing
…	

sorted(zip(temp…,cities))

26,	Beijing
29,	Chengdu
…	

```
s_cities = [ x for _, x in
sorted(zip(temperatures, cities))]
```

sorted_demo.py

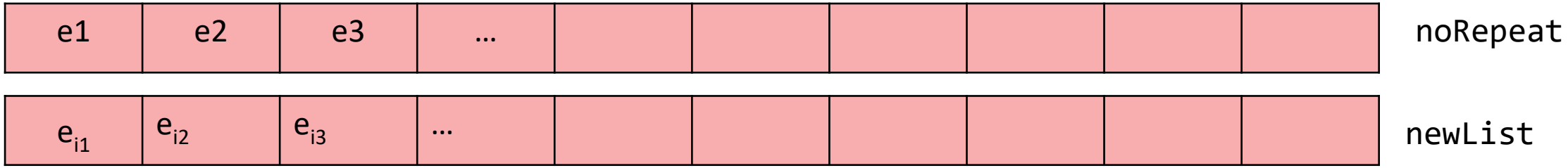
```
>>> cities = ['Shanghai', 'Beijing', 'Wuhan', 'Chengdu', 'Nanjing']
>>> temperatures = [30, 26, 34, 29, 38]
>>> pairs = sorted(zip(temperatures, cities))
>>> pairs
[(26, 'Beijing'), (29, 'Chengdu'), (30, 'Shanghai'), (34, 'Wuhan'), (38, 'Nanjing')]
>>> [item[1] for item in pairs]
['Beijing', 'Chengdu', 'Shanghai', 'Wuhan', 'Nanjing']
```

内置函数sorted的实践应用

利用**set不重复特性可去除重复元素**，但是无序特性导致产生的list不是按照原来的顺序

```
import random
# 产生100个[0,10000)之间的随机整数，保存在列表中
listRandom = [random.choice(range(10000)) for i in
range(100)]
noRepeat = list(set(listRandom))
newList = sorted(noRepeat, key=listRandom.index)
# 如果原地排序，则
# noRepeat.sort(key=listRandom.index)
```

```
>>> s = [0, 11, 8, 7, 9, 10, 2, 3, 3, 13]
>>> list(set(s))
[0, 2, 3, 7, 8, 9, 10, 11, 13]
>>> s.index
<built-in method index of list object at
0x0568A058>
>>> sorted(set(s), key=s.index)
[0, 11, 8, 7, 9, 10, 2, 3, 13]
```



sorted(noRepeat, key=listRandom.index)

listRandom.index (e_{i1})<=listRandom.index (e_{i1})<=listRandom.index (e_{i1})<=…

value在listRandom中的下标顺序

用于序列操作的常用内置函数

- len(seq)：返回容器对象seq中的元素个数，适用于列表、range、元组、字典、集合、字符串等
- max和min：两种语法
  - max(arg1, arg2, *args, key=func)、min(arg1, arg2, *args, key=func): 传递的参数中的最大或最小值
  - max(iterable, default=obj, key=func)、 min(iterable default=obj, key=func)：返回迭代对象中的最大或最小**元素**，如果为空，返回obj
  - **key指出如何比较大小，即基于调用函数key(item)的大小来判断。缺省比较参数或者元素的大小**
  - 传递的参数或者可迭代对象的元素并不要求一定是数值类型，只要可以按照相应的基准比较大小即可
- sum(iterable, start=0)：对**数值型可迭代对象**的元素进行求和运算，最后加上start（缺省为0）。元素为非数值型时抛出异常TypeError

```
>>> s = [1, 4, -5, -7, 6]
>>> len(s)
5
>>> max(1, 4, -5, -7, 6)
6
>>> min(1, 4, -5, -7, 6)
-7
>>> max(s)
6
>>> min(s)
-7
>>> max(1, 4, -5, -7, 6, key=abs)
-7
>>> min(1, 4, -5, -7, 6, key=abs)
1
>>> max(s, key=abs)
-7
>>> words=['nice','to', 'console']
>>> max(words, key=len)
'console'
>>> max([1,2], [2,4])
[2, 4]
>>> sum(s)
-1
```