## 内置序列类型概览

**容器序列**

list、tuple和collections.deque这些序列能存放不同类型的数据。

**扁平序列**

str、bytes、bytearray、memoryview和array.array，这些序列只能容纳一种类型。

容器序列存放的是它们所包含的**任意类型的对象的引用，而扁平序列里存放的是值而不是引用**。换句话说，扁平序列其实是一段连续的内存空间。由此可见扁平序列其实更加紧

凑，但是它里面只能存放诸如字符、字节和数值这种基础类型。

**可变序列**

list、bytearray、array.array、collections.deque和memoryview

**不可变序列**

tuple、str和bytes。

## 字典

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

# 创建字典

# 创建一个空字典

dict1 = {}

# 创建非空字典

dict1 = {*"name"*: *'Zara'*, *'age'*: 17}

# 通过赋值创建字典

dict1[*'class'*] = *'first'*

dict1[*'sex'*] = *'W'*

print dict1 # {'age': 17, 'sex': 'W', 'name': 'Zara', 'class': 'first'}

# 通过赋值创建字典

dict1 = dict(name=*'Zara'*, age = 17)

print dict1 # 'age': 17, 'name': 'Zara'}

# 通过zip函数创建字典

dict1 = dict(zip([*'name'*, *'age'*], [*'Zara'*, 17])) # 等价于dict([('name', 'Zara'), ('zge', 17)])

print dict1 # {'age': 17, 'name': 'Zara'}

# 通过formkeys函数构造含有默认值的字典

dict1 = dict.fromkeys([*'name'*, *'age'*])

print dict1 # {'age': None, 'name': None}

# 通过dict.fromkeys()方法设置默认值

dict1 = dict.fromkeys([*'id'*, *'num'*], 100)

print dict1 # {'num': 100, 'id': 100}

# 删除字典中所有项

del dict1[*'id'*] # 删除字典中id键值对,{'num': 100}

print dict1

# 清空字典中所有键值对

dict1.clear()

print dict1 # {}

# 返回字典的浅拷贝

dict1 = {*"name"*: *'Zara'*, *'age'*: 17}

print dict1.copy()

# 获取key的值，不存在则返回默认值

print dict1.get(*'class'*, []) # 返回字典中键对应的值，不存在则返回默认值

print dict1.items() # 以列表的形式返回字典中的键值对

print dict1.values() # 返回字典中的键值列表

print dict1.keys() # 返回字典中的value值列表

print dict1.has\_key(*'name'*) # 判断字典中键是否存在

dict1.update({*'sex'*: *'male'*, *'age'*: 25})

print dict1 # 更新字典

# 设置对应键值的默认值，如果对应键的值已经存在，则不设置

dict1.setdefault(*'age'*, [])

dict1.setdefault(*'class'*, *'first'*)

print dict1

# 弹出对应键的值，不存在则抛出keyerror

print dict1.pop(*'age'*)

# 随机返回字典的一对键值对元组

print dict1.popitem()

# 字典解析

D = {x: x\*x for x in range(10)}

print D # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

>>> D = {idx: x for idx, x in enumerate(open(r'C:\E\test\test.txt'))}

>>> D

{0: 'Hello\n', 1: '6\n'}

## 列表

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

list1 = [1, 2, 3, 4, 5, 6]

list2 = [*'a'*, *'b'*, *'c'*, *'d'*]

list3 = [*'physics'*, *'chemistry'*, 1997, 2000]

# 访问列表中的值

print *"list1[1:3]:"*, list1[1:3]

# 更新列表

list3[2] = 2001

print *"list3:"*, list3

# 删除列表中的元素

del list3[2]

print *"list3:"*, list3

# 列表的脚本操作符

print *"list1 + list2:"*, list1+list2

print *"list1 \* 4"*, list1\*4

print *"3 in list1:"*, 3 in list1

list6 = [1, 2, 3, 4]

print *"id(list6)=%d"* % id(list6)

print *"id(list6+=[1, 2]=%d)"* % id(list6) # +=在原地扩展，+生成新的列表对象

# 列表截取

print *"list1[0:1000]:"*, list1[0:1000] # 超出index范围列表不会抛出异常

print *"list1[-3::2]:"*, list1[-10::2] # 超出范围从index为0开始解析

print *"list1[::-1]:"*, list1[::-1] # 列表逆序

print *"list1[::-2]:"*, list1[::-2] # 列表以2间隔逆序

# 列表插入

list1[2:4] = [1, 2, 3, 4]

print *"list1:"*, list1

list2[1:4] = [1]

print *"list2:"*, list2

# 常见函数

print *"cmp(%s, %s):%d"* % (list1, list2, cmp(list1, list2)) # 比较函数

print *"len(%s)=%d"* % (list1, len(list1)) # 列表的长度

print *"max(%s)=%d, min(%s)=%d"* % (list1, max(list1), list2, min(list2))

print *"list({'a': 1, 'b': 2}):"*, list({*'a'*: 1, *'b'*: 2}) # 将一个序列转化为列表

# 列表的常见方法

list1.append(100) # 列表的末尾添加一个值

print *"list1.append(100):"*, list1

print *"list1.count(2):"*, list1.count(2) # 统计列表中元素2出现的次数

print *"id(list1):"*, id(list1)

print *"%s.extend(%s):"* % (list1, list2) # 在列表末尾一次性追加另一个序列中的多个值

#extend(...)

# L.extend(iterable) -- extend list by appending elements from the iterable

list1.extend(list2)

print *"list1=%s"* % list1

print *"id(list1):"*, id(list1)

list3.insert(10, *'abc'*) # 超出了索引范围也可以插入，直接插入末尾 ,index也可以是负数

print *"list3.insert(10, 'abc'):"*

print list3

list3.insert(-2, *'cba'*) # 也可以从后向前插入

print *"list3.insert(-2, 'cba'):"*

print list3

print *"list3.pop():%s list3.pop(-2):%s list3.pop(2):%s"* % (list3.pop(), list3.pop(-2), list3.pop(2)) # 从列表中移除一个元素

list4 = [1, 4, 2, 3, 2]

print *"list:"*, list4

list4.remove(2) # 删除列表中的第一个匹配项

print *"[1, 4, 2, 3, 2].remove(2):"*, list4

list4.reverse() # 列表原地反向

print *"list4.reverse():"*, list4

list5 = [1, -2, 3, -3, 6, 8, 9]

list5.sort(key=lambda x: x\*x, reverse=True) # 列表排序函数，原地修改

print list5

L = [[(x, y) for x in range(4)] for y in range(5)]

print L

>>> L

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

**列表删除陷阱**：

>>> L = [1, 2, "", ""]

>>> for i in L:

if i == "":

L.remove(i)

>>> L

[1, 2, ''] # ''并没有被删除

>>> for i in L:

print id(i)

30900104

30900080

30749000

30749000

>>> for i in L:

print i, id(i)

if i=="":

L.remove(i)

1 30900104

2 30900080

30749000

for in 是对下标进行操作，而remove是对值进行操作，当for到达索引为3，即第一个空字符位置时，符合条件即删除。因为list是线性表，所以删除这个空字符时，同时后面的所有元素自动移动位置1，此时L为[1,2,’’],因为for in 已经遍历到索引3处，所以循环结束。

解决方法：

>>> L = [i for i in L if i!= '']

>>> L

[1, 2]

>>> L = [1,2,'','']

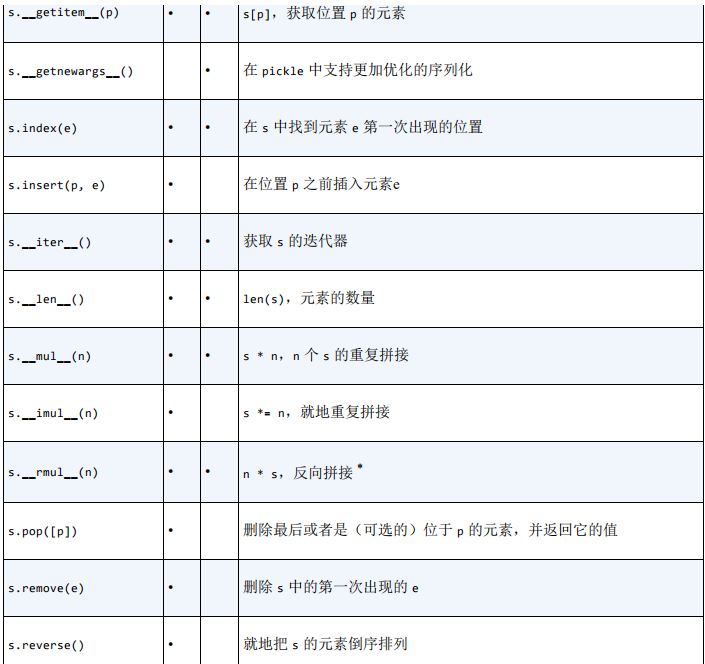
>>> while '' in L:

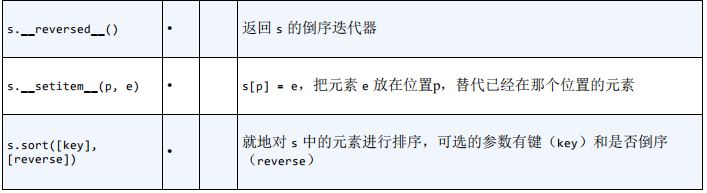
L.remove('')

>>> L

[1, 2]







## 元组

支持方法：

| count(...)

| T.count(value) -> integer -- return number of occurrences of value

|

| index(...)

| T.index(value, [start, [stop]]) -> integer -- return first index of value.

| Raises ValueError if the value is not present.

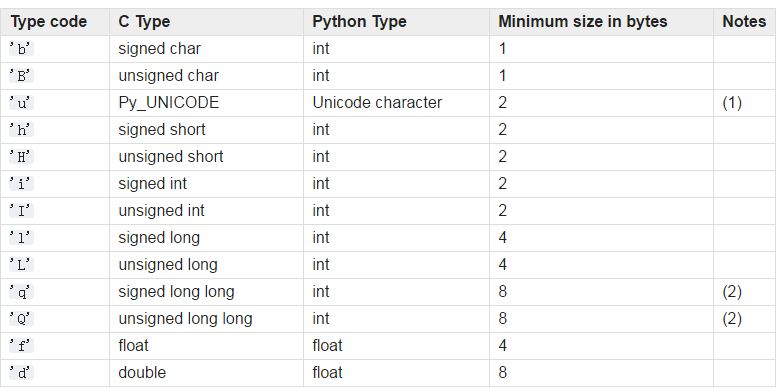
### 元组不仅仅是不可变得序列

元组其实是对数据的记录：元组中的每个元素都存放了记录中一个字段的数据，外加这个

字段的位置。

## 数组

如果我们需要一个只包含数字的列表，那么array.array比list更高效。



class array(\_\_builtin\_\_.object)

| array(typecode [, initializer]) -> array

|

| Return a new array whose items are restricted by typecode, and

| initialized from the optional initializer value, which must be a list,

| string or iterable over elements of the appropriate type.

|

| Arrays represent basic values and behave very much like lists, except

| the type of objects stored in them is constrained.

|

| Methods:

|

| append() -- append a new item to the end of the array

| buffer\_info() -- return information giving the current memory info

| byteswap() -- byteswap all the items of the array

| count() -- return number of occurrences of an object

| extend() -- extend array by appending multiple elements from an iterable

| fromfile() -- read items from a file object

| fromlist() -- append items from the list

| fromstring() -- append items from the string

| index() -- return index of first occurrence of an object

| insert() -- insert a new item into the array at a provided position

| pop() -- remove and return item (default last)

| read() -- DEPRECATED, use fromfile()

| remove() -- remove first occurrence of an object

| reverse() -- reverse the order of the items in the array

| tofile() -- write all items to a file object

| tolist() -- return the array converted to an ordinary list

| tostring() -- return the array converted to a string

| write() -- DEPRECATED, use tofile()

|

| Attributes:

|

| typecode -- the typecode character used to create the array

| itemsize -- the length in bytes of one array item

例：

>>> from array import array

>>> from random import random

>>> floats = array('d', (random() for i in range(10\*\*7)))

>>> floats[-1]

0.03206417003211881

>>> len(floats)

10000000

>>> fp = open(r'c:\e\dir0\floats.bin', 'wb')

>>> floats.tofile(fp)

>>> fp.close()

>>>

>>> floats2 = array('d')

>>> fp = open(r'c:\e\dir0\floats.bin', 'rb')

>>> floats2.fromfile(fp, 10\*\*7)

>>> floats2[-1]

0.03206417003211881

>>> floats2 == floats

True

>>> fp.close()

用 array.fromfile

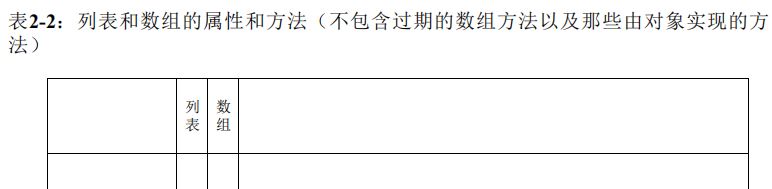
从一个二进制文件里读出 1000 万个双精度浮点数只需要 0.1 秒，这比从文本文件里读取

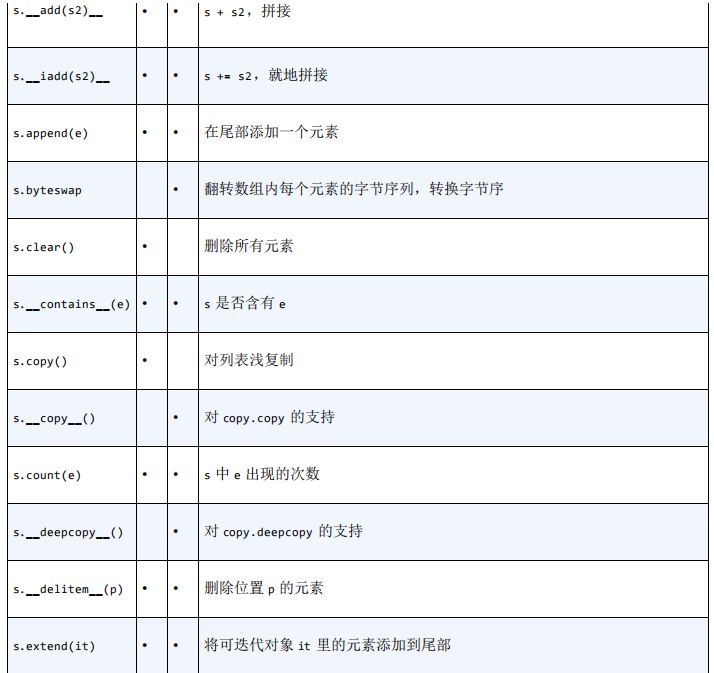
的速度要快 60 倍，因为后者会使用内置的 float方法把每一行文字转换为浮点数。另

外，使用 array.tofile 写入到二进制文件，比以每行一个浮点数的方式把所有数字写入到文本文件要快 7 倍。另外，1000 万个这样的数在二进制文件里只占用 80 000 000 个

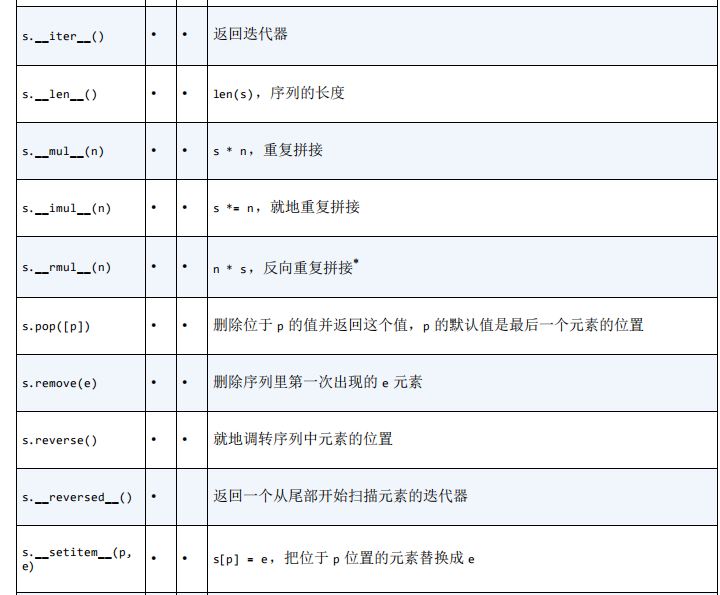
字节（每个浮点数占用 8 个字节，不需要任何额外空间），如果是文本文件的话，我们需

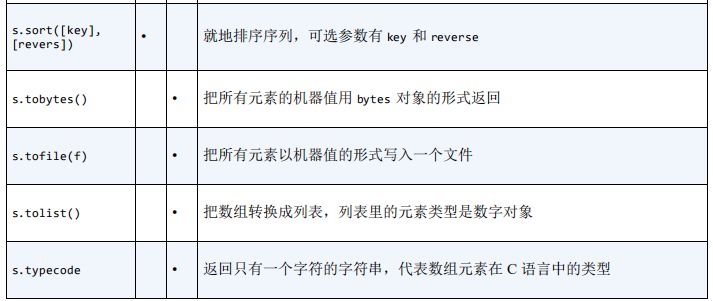
要 181 515 739 个字节。











从 Python 3.4 开始，数组类型不再支持诸如 list.sort() 这种就地排序方法。要给

数组排序的话，得用 sorted 函数新建一个数组：

a = array.array(a.typecode, sorted(a))

## 内存视图

**memoryview()**函数返回给定参数的内存查看对象(Momory view)。

所谓内存查看对象，是指对支持缓冲区协议的数据进行包装，在不需要复制对象基础上允许Python代码访问。

**python3.x**

cast(self, /, format, \*, shape)

Cast a memoryview to a new format or shape.

>>> from array import array

>>>

>>> numbers = array('h', range(-2, 3))

>>> memv = memoryview(numbers)

>>> len(memv)

5

>>> memv[0]

-2

>>> memv\_oct = memv.cast('B')

>>> memv\_oct

<memory at 0x0000000002FB7288>

>>> memv\_oct.tolist()

[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]

>>> memv\_oct

<memory at 0x0000000002FB7288>

>>> memv\_oct[5]

0

>>> memv\_oct[5] = 4

>>> numbers

array('h', [-2, -1, 1024, 1, 2])

1. 利用含有 5 个短整型有符号整数的数组（类型码是 'h'）创建一个 memoryview。
2. memv 里的 5 个元素跟数组里的没有区别。
3. 创建一个 memv\_oct，这一次是把 memv 里的内容转换成 'B' 类型，也就是无符号字符。
4. 以列表的形式查看 memv\_oct 的内容。
5. 把位于位置 5 的字节赋值成 4。
6. 因为我们把占 2 个字节的整数的高位字节改成了 4，所以这个有符号整数的值就变成了1024。