



武汉大学  
WUHAN UNIVERSITY

# 数据结构与算法

## 基于数组的序列

---

张晓平

武汉大学数学与统计学院

# Table of contents

1. Python 的序列类型
2. 低级数组 (low-level's array)
3. 动态数组 (Dynamic Arrays)
4. Python 序列类型的效率
5. 基于数组的序列：应用

# Python 的序列类型

# Python 的序列类型

本章主要考察 Python 的序列 (Sequence) 类，即内置的列表 (list)，元组 (tuple) 和字符串 (str) 类.

# Python 的序列类型

本章主要考察 Python 的序列 (Sequence) 类，即内置的列表 (list)，元组 (tuple) 和字符串 (str) 类。

## 相同点

- 支持索引来访问元素，如 `seq[k]`
- 使用 `数组 (array)` 来表示序列。

# Python 的序列类型

本章主要考察 Python 的序列 (Sequence) 类，即内置的列表 (list)，元组 (tuple) 和字符串 (str) 类。

## 相同点

- 支持索引来访问元素，如 `seq[k]`
- 使用 `数组 (array)` 来表示序列。

## 不同点

- 抽象
- 在 Python 内部表示的方式

# Python 的序列类型

本章主要考察 Python 的序列 (Sequence) 类，即内置的列表 (list)，元组 (tuple) 和字符串 (str) 类。

## 相同点

- 支持索引来访问元素，如 `seq[k]`
- 使用 `数组 (array)` 来表示序列。

## 不同点

- 抽象
- 在 Python 内部表示的方式

这些类在 Python 程序中的使用非常广泛。基于这些类我们开发更复杂的数据结构，因此必须弄清楚它们的公共行为和内部工作机制。

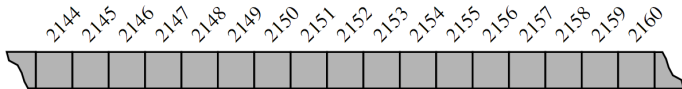
**低级数组 (low-level's array)**



## 低级数组 (low-level's array)

### 内存地址

内存的每个字节都与作为其地址的唯一数字相关联。



**Figure 5.1:** A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

内存地址通常与内存系统的物理布局相协调，故常以[连续方式](#)来表示这些数字。

## 低级数组 (low-level's array)

### 定义：数组

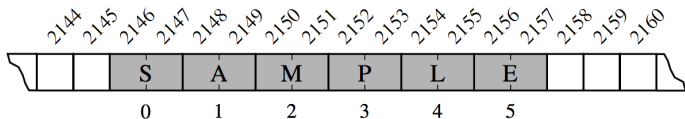
可以将一组相关变量一个接一个地存储在内存的连续区域中，这样的表示称为数组。

## 低级数组 (low-level's array)

### 例

文本字符串以字符的有序序列存储。如下图中的'SAMPLE', 将其描述为一个 6 个字符的数组, 尽管它需要 12 个字节的内存。

将数组中的每个位置称为一个**单元格 (cell)**, 并使用**整数索引**来描述其在数组中的位置, 单元格的编号为 0, 1, 2, ...。



**Figure 5.2:** A Python string embedded as an array of characters in the computer's memory. We assume that **each Unicode character of the string requires two bytes of memory**. The numbers below the entries are indices into the string.

## 低级数组 (low-level's array)

### 注

数组的每个单元格必须使用相同的字节数。这一要求允许根据数组的索引在恒定时间内访问数组的任意单元格。

### 例

给定数组开始的内存地址  $start$ 、每个元素的字节数  $cellsize$  以及数组中的一个给定索引  $index$ ，则给定索引的内存地址为  $start + cellsize * index$ .

**低级数组 (low-level's array)**

**引用数组 (Referential Arrays)**

## 引用数组 (Referential Arrays)

再次强调，Python 数组的每个单元格必须使用相同的字节数.

## 引用数组 (Referential Arrays)

再次强调，Python 数组的每个单元格必须使用相同的字节数。

### 问题

给定一个列表

```
['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen',  
, 'Virginia']
```

如何在数组中表示这样的列表呢？

## 引用数组 (Referential Arrays)

方法 1: 为每个单元格保留足够的空间来存储最长字符串, 但这非常浪费。



## 引用数组 (Referential Arrays)

方法 1: 为每个单元格保留足够的空间来存储最长字符串, 但这非常浪费。

方法 2: Python 使用对象引用数组的内部存储机制表示列表或元组实例。在最底层, 存储的是序列元素所在的连续内存地址序列。

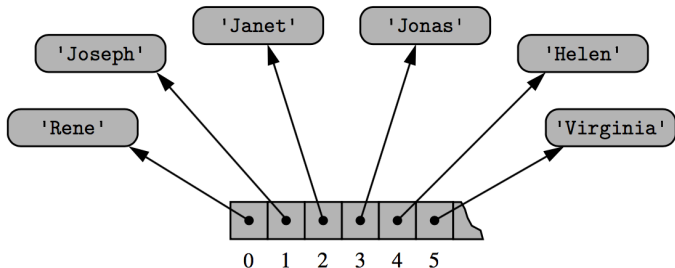


Figure 5.4: An array storing references to strings.

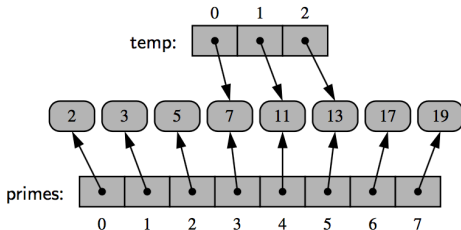
## 引用数组 (Referential Arrays)

尽管各元素的相对大小可能不同，但用于存储每个元素内存地址的位数 (bits) 是固定的。通过这种方式，Python 可以基于其索引支持对列表或元组元素的恒定时间访问。

## 引用数组 (Referential Arrays)

例

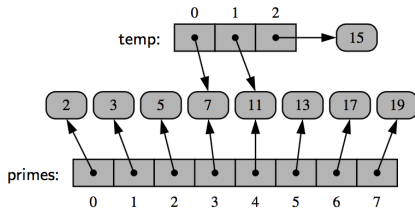
计算列表的一个切片时，返回的是一个新的列表实例，但这个新列表引用了原始列表中的相同元素。



**Figure 5.5:** The result of the command `temp = primes[3:6]`.

## 引用数组 (Referential Arrays)

修改切片中某元素的值，只会改变该单元格的引用，其他单元格仍然引用原始列表中的元素。

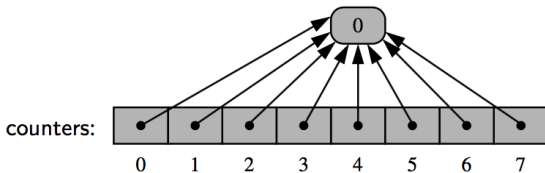


**Figure 5.6:** The result of the command `temp[2] = 15` upon the configuration portrayed in Figure 5.5.

## 引用数组 (Referential Arrays)

```
data = [0] * 8
```

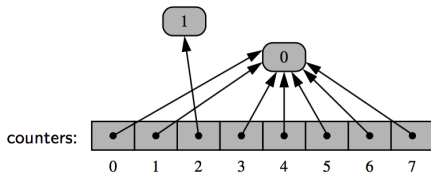
生成一个长度为 8 的列表，且所有这 8 个元素的值皆为 0。本质上，列表中的 8 个单元格都引用同一个对象。



**Figure 5.7:** The result of the command `data = [0] * 8`.

## 引用数组 (Referential Arrays)

```
data[2] += 1
```



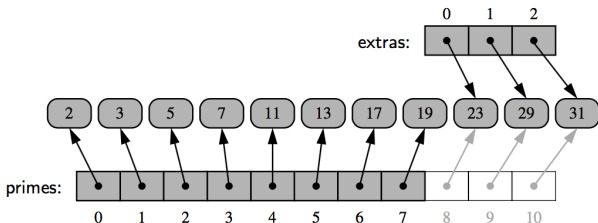
**Figure 5.8:** The result of command `data[2] += 1` upon the list from Figure 5.7.

索引为 2 的单元格的引用发生了变化，其余单元格的引用不受影响。

## 引用数组 (Referential Arrays)

```
>>> primes = [2,3,5,7,11,13,17,19]
>>> extras = [23,29,31]
>>> primes.extend(extras)
```

`extend` 命令用于将一个列表中的所有元素添加到另一个列表的末尾。

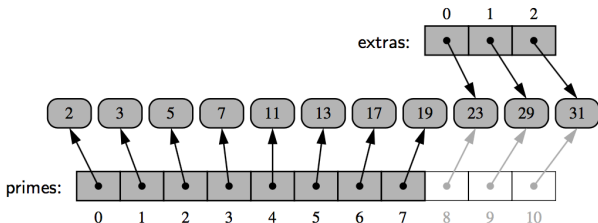


**Figure 5.9:** The effect of command `primes.extend(extras)`, shown in light gray.

## 引用数组 (Referential Arrays)

```
>>> primes = [2,3,5,7,11,13,17,19]
>>> extras = [23,29,31]
>>> primes.extend(extras)
```

`extend` 命令用于将一个列表中的所有元素添加到另一个列表的末尾。



**Figure 5.9:** The effect of command `primes.extend(extras)`, shown in light gray.

扩展列表不接收这些元素的副本，而是接收对这些元素的引用。



**低级数组 (low-level's array)**

**Python 中的紧凑数组 (Compact Arrays in Python)**

# Python 中的紧凑数组 (Compact Arrays in Python)

字符串使用字符数组表示，而非引用数组。

这种更直接的表示形式被称为**紧凑数组**，因为数组存储的是原始数据 (对于字符串就是字符)。

S	A	M	P	L	E
0	1	2	3	4	5

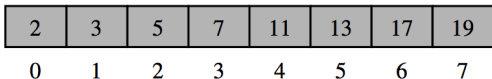
## Python 中的紧凑数组 (Compact Arrays in Python)

在计算性能方面，相比于引用结构，紧凑数组有如下几个优点：

- 总的内存使用率会低得多；
- 原始数据连续存储在内存中。

## Python 中的紧凑数组 (Compact Arrays in Python)

紧凑数组的主要支持位于名为 `array` 的模块中。该模块定义了一个类，也称为 `array`，为原始数据类型的数组提供紧凑存储。



The diagram illustrates a compact array structure. It consists of a horizontal row of eight gray rectangular boxes, each containing an integer value. Below each box is its corresponding index, ranging from 0 to 7. The values are 2, 3, 5, 7, 11, 13, 17, and 19, respectively.

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

**Figure 5.10:** Integers stored compactly as elements of a Python array.

## Python 中的紧凑数组 (Compact Arrays in Python)

```
>>> from array import array
>>> print(array('i', [1,2,3,4,5]))
array('i', [1, 2, 3, 4])
>>> print(array('f', [1,2,3,4,5]))
array('f', [1.0, 2.0, 3.0, 4.0])
>>> print(array('d', [1,2,3,4,5]))
array('d', [1.0, 2.0, 3.0, 4.0])
```

## Python 中的紧凑数组 (Compact Arrays in Python)

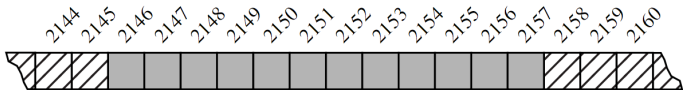
Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	double	8

**Table 5.1:** Type codes supported by the array module.

## 动态数组 (Dynamic Arrays)

## 动态数组 (Dynamic Arrays)

创建低级数组时，必须显式地声明该数组的精确大小，以便系统为其存储正确分配连续的内存。



**Figure 5.11:** An array of 12 bytes allocated in memory locations 2146 through 2157.



## 动态数组 (Dynamic Arrays)

由于系统可能会指定相邻的内存位置来存储其他数据，因此不能通过扩展到后续单元来简单地增加数组的容量。注意，在表示 Python 元组或 str 实例时，这个约束没有问题，因为它们是不可变对象

## 动态数组 (Dynamic Arrays)

Python 的 list 类提供了一个更有趣的抽象。虽然列表在构造时有一个指定长度，但是类允许我们向列表中添加元素，并且对列表的总容量没有明显的限制。为提供这种抽象，Python 依赖于一种称为“**动态数组**”的算法技巧。

## 动态数组 (Dynamic Arrays)

列表实例维护一个底层数组，该数组的容量通常大于当前列表长度。

有了这个额外的容量，通过使用数组的下一个可用单元格，就可以很容易将新元素附加到列表中。

## 动态数组 (Dynamic Arrays)

如果用户继续将元素追加到列表中，则预留的容量会最终耗尽。

- 在此情况下，类会从系统请求一个新的更大的数组，并初始化新的数组，使得原数组中的元素依次存入新数组的靠前位置上。
- 然后原数组不再需要，系统将其回收。

## 动态数组 (Dynamic Arrays)

如果用户继续将元素追加到列表中，则预留的容量会最终耗尽。

- 在此情况下，类会从系统请求一个新的更大的数组，并初始化新的数组，使得原数组中的元素依次存入新数组的靠前位置上。
- 然后原数组不再需要，系统将其回收。

非常像寄居蟹 (hermit crab)

## 动态数组 (Dynamic Arrays)

```
import sys
try:
    n = int(sys.argv[1])
except:
    n = 100
data = []
for k in range(n):
    a = len(data)
    b = sys.getsizeof(data)
    print(f'Length: {a:3d}; Size in bytes: {b:4d} ')
    data.append(None)
```

## 动态数组 (Dynamic Arrays)

```
$ python listsize.py 6
Length:    0; Size in bytes:    64
Length:    1; Size in bytes:    96
Length:    2; Size in bytes:    96
Length:    3; Size in bytes:    96
Length:    4; Size in bytes:    96
Length:    5; Size in bytes:   128
```

# 动态数组 (Dynamic Arrays)

动态数组的实现



尽管 Python 的 `list` 类提供了动态数组的高度优化实现，但是看看它是如何实现的仍具有指导意义。

尽管 Python 的 `list` 类提供了动态数组的高度优化实现，但是看看它是如何实现的仍具有指导意义。

实现动态数组的关键是提供一种方法来扩展存储列表元素的数组  $A$ 。

若底层数组已满，欲将元素追加到列表，将执行以下步骤：

- 分配具有更大容量的数组  $B$ ；
- 令  $B[i] = A[i]$ ,  $i = 0, \dots, n-1$ ，其中  $n$  表示当前项目数；
- 令  $A = B$ ，此后使用  $B$  作为支持列表的数组。

若底层数组已满，欲将元素追加到列表，将执行以下步骤：

- 分配具有更大容量的数组  $B$ ；
- 令  $B[i] = A[i]$ ,  $i = 0, \dots, n-1$ ，其中  $n$  表示当前项目数；
- 令  $A = B$ ，此后使用  $B$  作为支持列表的数组。

新数组的容量应该多大呢？。

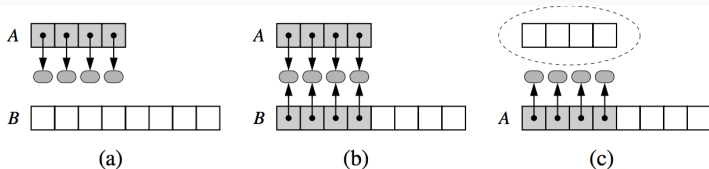
若底层数组已满，欲将元素追加到列表，将执行以下步骤：

- 分配具有更大容量的数组  $B$ ；
- 令  $B[i] = A[i]$ ,  $i = 0, \dots, n-1$ ，其中  $n$  表示当前项目数；
- 令  $A = B$ ，此后使用  $B$  作为支持列表的数组。

新数组的容量应该多大呢？。

一个常用的规则是**新数组的容量是已经填充的现有数组的两倍。**

# 动态数组的实现



**Figure 5.12:** An illustration of the three steps for “growing” a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. **Not shown is the future garbage collection of the old array, or the insertion of the new element.**

## 动态数组的实现

```
from ctypes import py_object
class DynamicArray(object):
    def __init__(self):
        self._n = 0
        self._capacity = 1
        self._A = self._make_array(self._capacity)

    def _make_array(self, c):
        return (c * py_object)()
```

## 动态数组的实现

```
def __len__(self):  
    return self._n  
  
def __getitem__(self, k):  
    if not 0 <= k < self._n:  
        raise IndexError('invalid index')  
    return self._A[k]
```



## 动态数组的实现

```
def append(self, obj):
    if self._n == self._capacity:
        self._resize(2 * self._capacity)
    self._A[self._n] = obj
    self._n += 1

def _resize(self, c):
    B = self._make_array(c)
    for k in range(self._n):
        B[k] = self._A[k]
    self._A = B
    self._capacity = c
```

## 动态数组的实现

```
def insert(self, k, value):
    if self._n == self._capacity:
        self._resize(2 * self._capacity)
    for j in range(self._n, k, -1):
        self._A[j] = self._A[j-1]
    self._A[k] = value
    self._n += 1

def remove(self, value):
    for k in range(self._n):
        if self._A[k] == value:
            for j in range(k, self._n - 1):
                self._A[j] = self._A[j+1]
            self._A[self._n - 1] = None
            self._n -= 1
    return
    raise ValueError('value not found')
```

## 动态数组的实现

```
def pop(self, k=None):
    if k == None:
        value = self._A[self._n-1]
        self._A[self._n-1] = None
        self._n -= 1
        return value
    else:
        if not 0 <= k < self._n:
            raise IndexError('invalid index')
        value = self._A[k]
        for j in range(k, self._n-1):
            self._A[j] = self._A[j+1]
        self._A[self._n-1] = None
        self._n -= 1
        return value
```

## 动态数组的实现

```
def __repr__(self):  
    return f'{self._A[0:self._n]}'
```

## 动态数组的实现

```
if __name__ == '__main__':  
    a = DynamicArray()  
    a.append('1')  
    a.append('2')  
    a.insert(1, 'abc')  
    a.append('2')  
    print(a)  
  
    a.remove('2')  
    print(a)  
  
    a.pop(1)  
    print(a)  
  
    a.pop()  
    print(a)
```

## 动态数组的实现

```
['1', 'abc', '2', '2']
```

```
['1', 'abc', '2']
```

```
['1', '2']
```

```
['1']
```

## Python 序列类型的效率

# Python 序列类型的效率

Python 的列表和元组类



Operation	Running Time
<code>len(data)</code>	$O(1)$
<code>data[j]</code>	$O(1)$
<code>data.count(value)</code>	$O(n)$
<code>data.index(value)</code>	$O(k+1)$
<code>value in data</code>	$O(k+1)$
<code>data1 == data2</code> (similarly <code>!=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> )	$O(k+1)$
<code>data[j:k]</code>	$O(k-j+1)$
<code>data1 + data2</code>	$O(n_1 + n_2)$
<code>c * data</code>	$O(cn)$

**Table 5.3:** Asymptotic performance of the nonmutating behaviors of the list and tuple classes. Identifiers `data`, `data1`, and `data2` designate instances of the list or tuple class, and  $n$ ,  $n_1$ , and  $n_2$  their respective lengths. For the containment check and index method,  $k$  represents the index of the leftmost occurrence (with  $k = n$  if there is no occurrence). For comparisons between two sequences, we let  $k$  denote the leftmost index at which they disagree or else  $k = \min(n_1, n_2)$ .

- `len(data)`
- `data[j]`

- `count`

计算计数的循环必须贯穿整个序列

- `index, __contains__`

一旦发现元素的索引或确定元素的索引，一旦找到了期望值的最左边出现，则循环立即退出。

## 例

```
data = list(range(10000000))
```

- `5 in data`: Best
- `9999995 in data`: Middle
- `-5 in data`: Worst

## 创建新实例 (Creating New Instances)

渐近行为与结果的长度成正比。

### 例

- 切片 `data[6000000:6000008]` 几乎可以立即构造，因为它只有八个元素；
- 切片 `data[6000000:7000000]` 有 100 万个元素，因此创建比较耗时。

Operation	Running Time
<code>data[j] = val</code>	$O(1)$
<code>data.append(value)</code>	$O(1)^*$
<code>data.insert(k, value)</code>	$O(n - k + 1)^*$
<code>data.pop()</code>	$O(1)^*$
<code>data.pop(k)</code> <code>del data[k]</code>	$O(n - k)^*$
<code>data.remove(value)</code>	$O(n)^*$
<code>data1.extend(data2)</code> <code>data1 += data2</code>	$O(n_2)^*$
<code>data.reverse()</code>	$O(n)$
<code>data.sort()</code>	$O(n \log n)$

\*amortized

**Table 5.4:** Asymptotic performance of the mutating behaviors of the list class. Identifiers `data`, `data1`, and `data2` designate instances of the list class, and  $n$ ,  $n_1$ , and  $n_2$  their respective lengths.

最简单的可变行为是`data[j] = val`，并由特殊方法`__setitem__`支持。该操作的时间复杂度为  $O(1)$ ，因为

- 它只是用一个新值替换列表中的一个元素
- 其他元素不受影响，底层数组的大小也不会改变。

## 向列表中添加元素

- `append`方法

由于底层数组已调整大小，其时间复杂度为  $O(n)$ ，但在摊余意义上为  $O(1)$ .



## 向列表中添加元素

- `insert`方法

`insert(k, value)` 将给定值插入到索引  $0 \leq k \leq n$  处，同时将所有后续元素向右移动以腾出空间。

## Adding Elements to a List

```
def insert(self, k, value):  
    if self._n == self._capacity:  
        self._resize(2 * self._capacity)  
    for j in range(self._n, k, -1):  
        self._A[j] = self._A[j-1]  
    self._A[k] = value  
    self._n += 1
```

# Adding Elements to a List

	$N$				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

**Table 5.5:** Average running time of `insert(k, val)`, measured in microseconds, as observed over a sequence of  $N$  calls, starting with an empty list. We let  $n$  denote the size of the current list (as opposed to the final list).

# Adding Elements to a List

	$N$				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

**Table 5.5:** Average running time of `insert(k, val)`, measured in microseconds, as observed over a sequence of  $N$  calls, starting with an empty list. We let  $n$  denote the size of the current list (as opposed to the final list).

- 在列表的开头插入是最昂贵的，每次操作需要线性时间；
- 中间插入需要大约一半的时间，但仍然是  $O(n)$  时间；
- 在末尾插入为  $O(1)$  时间，类似于 `append` 方法。

# 从列表中删除元素

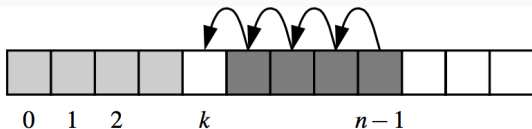
- `pop()`: 从列表中删除最后一个元素

这是最有效的，因为所有其他元素都保留在其原始位置。这实际上是一个  $O(1)$  操作，不过 Python 偶尔会收缩底层动态数组以节省内存。

## 从列表中删除元素

- `pop(k)`: 移除位于列表中索引  $k < n$  处的元素，将所有后续元素向左移动以填补移除所产生的空白。

此操作的效率为  $O(n - k)$ ，因为移位量取决于索引  $k$  的选择。



**Figure 5.17:** Removing an element at index  $k$  of a dynamic array.

# 从列表中删除元素

- `remove`方法

`remove(value)` 允许调用方指定要删除的值。

此操作的效率为  $O(n)$ 。流程的一部分从开始搜索到在索引  $k$  处找到值，而其余部分从  $k$  到结束迭代，以便将元素向左移动。

## 从列表中删除元素

```
def remove(self, value):  
    for k in range(self._n):  
        if self._A[k] == value:  
            for j in range(k, self._n - 1):  
                self._A[j] = self._A[j+1]  
            self._A[self._n - 1] = None  
            self._n -= 1  
            return  
    raise ValueError('value not found')
```



- `extend`方法

将一个列表的所有元素添加到第二个列表的末尾

调用`data.extend(other)`产生与以下代码相同的结果:

```
for element in other:  
    data.append(element)
```

# 扩展列表

- `extend`方法

将一个列表的所有元素添加到第二个列表的末尾

调用`data.extend(other)`产生与以下代码相同的结果:

```
for element in other:  
    data.append(element)
```

在这两种情况下，运行时间都与另一个列表的长度成正比，并且是摊余的，因为第一个列表的底层数组可能会调整大小以容纳其他元素。

# 扩展列表

- `extend`方法

将一个列表的所有元素添加到第二个列表的末尾

调用`data.extend(other)`产生与以下代码相同的结果:

```
for element in other:  
    data.append(element)
```

在这两种情况下，运行时间都与另一个列表的长度成正比，并且是摊余的，因为第一个列表的底层数组可能会调整大小以容纳其他元素。

在实际应用中，由于渐近分析中隐藏的常数因子明显较小，因此扩展方法优于重复调用追加。

扩展的高效性源于三个方面：

1. 使用合适的 Python 方法总会有一些好处，因为这些方法通常是用编译语言（而不是解释的 python 代码）实现的。
2. 与多次调用一些单独的函数相比，调用单个函数以完成所有工作的开销更小。
3. 扩展效率的提高源于这样一个事实：更新后的列表的结果大小可以提前计算。如果第二个数据集相当大，则在使用重复的追加调用时，可能会多次调整底层动态数组的大小。使用一个扩展调用，最多将执行一个调整大小操作。

# 构造新列表

- 列表推导式

```
squares = [k*k for k in range(1, n+1)]
```

- 循环

```
squares = []  
for k in range(1, n+1):  
    squares.append(k*k)
```

列表推导式比通过重复追加来构建列表要快得多。

## 构造新列表

使用乘法运算符初始化常量值列表，如 `[0]*n`，以生成长度为  $n$  且所有值都等于零的列表。它比增量构建这样一个列表更有效。

## 基于数组的序列：应用

## 基于数组的序列：应用

### 记分牌 (score board)



### 问题

为保持一个高分序列，我们设计一个名为 `ScoreBoard` 的类。一个记分牌只能存储一定数量的高分。若记分牌已满，则只有当新的分数高于记分牌中的最低分时才能进入记分牌。记分牌的容量取决于游戏，如被设置为 10、50、500 等，正因如此，我们会在 `ScoreBoard` 的构造器中指定一个容量的参数。

## 记分牌 (score board)

### 例

假设已有一个容量为 7 的记分牌，

```
[('Mike', 1105), ('Rob', 750), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

# 记分牌 (score board)

## 例

假设已有一个容量为 7 的记分牌，

```
[('Mike', 1105), ('Rob', 750), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

- 有一个选手 Jill，其得分为 740，记分牌将更新为

```
[('Mike', 1105), ('Rob', 750), ('Jill', 740), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

# 记分牌 (score board)

## 例

假设已有一个容量为 7 的记分牌，

```
[('Mike', 1105), ('Rob', 750), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

- 有一个选手 Jill，其得分为 740，记分牌将更新为

```
[('Mike', 1105), ('Rob', 750), ('Jill', 740), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

- 又有一个选手 Jane，得分为 500，则她不能进入记分牌，记分牌不更新。

# 记分牌 (score board)

## 例

假设已有一个容量为 7 的记分牌，

```
[('Mike', 1105), ('Rob', 750), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

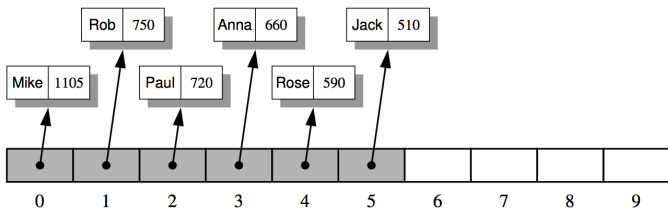
- 有一个选手 Jill，其得分为 740，记分牌将更新为

```
[('Mike', 1105), ('Rob', 750), ('Jill', 740), ('Paul', 720), ('Anna', 660), ('Rose', 590), ('Jack', 510)]
```

- 又有一个选手 Jane，得分为 500，则她不能进入记分牌，记分牌不更新。
- 再有一个选手 David，得分为 600，则记分牌将更新为

```
[('Mike', 1105), ('Rob', 750), ('Jill', 740), ('Paul', 720), ('Anna', 660), ('David', 600), ('Rose', 590)]
```

## 记分牌 (score board)



**Figure 5.18:** An illustration of an ordered list of length ten, storing references to six GameEntry objects in the cells from index 0 to 5, with the rest being None.

## 记分牌 (score board)

首先构造用于描述选手信息的类 `GameEntry`，其中包括选手姓名和分数：

```
class GameEntry(object):
    def __init__(self, name, score):
        self._name = name
        self._score = score
    def get_name(self):
        return self._name
    def get_score(self):
        return self._score
    def __str__(self):
        return f'({self._name}, {self._score})'
```

## 记分牌 (score board)

然后构造描述记分牌的 ScoreBoard 类，其中包含 add 方法，用于添加新进选手的信息：

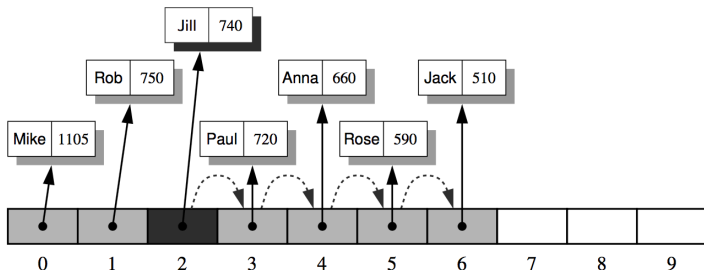
```
class ScoreBoard(object):  
    def __init__(self, capacity=5):  
        self._board = [None] * capacity  
        self._n = 0  
    def __getitem__(self, k):  
        return self._board[k]  
    def __str__(self):  
        return '\n'.join(str(self._board[j]) for j  
                           in range(self._n))
```



## 记分牌 (score board)

```
def add(self, entry):
    score = entry.get_score()
    good = self._n < len(self._board) or score >
        self._board[-1].get_score()
    if good:
        if self._n < len(self._board):
            self._n += 1
        j = self._n-1
        while j > 0 and self._board[j-1].get_score
            () < score:
            self._board[j] = self._board[j-1]
            j -= 1
        self._board[j] = entry
```

## 记分牌 (score board)



**Figure 5.19:** Adding a new GameEntry for Jill to the scoreboard. In order to make room for the new reference, we have to shift the references for game entries with smaller scores than the new one to the right by one cell. Then we can insert the new entry with index 2.

## 记分牌 (score board)

```
if __name__ == '__main__':
    board = ScoreBoard(5)
    for e in (
        ('Rob', 750), ('Mike', 1105), ('Rose', 590),
        ('Jill', 740), ('Jack', 510), ('Anna', 660),
        ('Paul', 720), ('Bob', 400),
    ):
        ge = GameEntry(e[0], e[1])
        board.add(ge)
    print(f'After considering {ge}, scoreboard
is:')
    print(board)
    print()
```

# 基于数组的序列：应用

## 插入排序

## 问题

对一个无序序列进行非递减排序。

# 插入排序

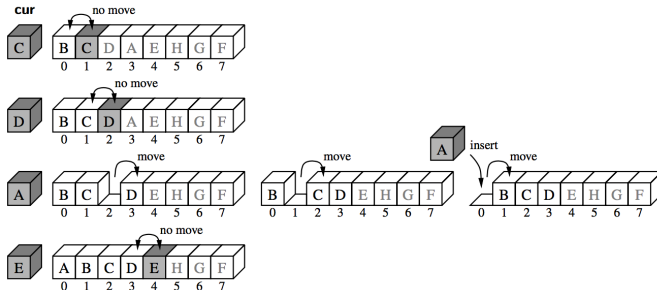
## 问题

对一个无序序列进行非递减排序。

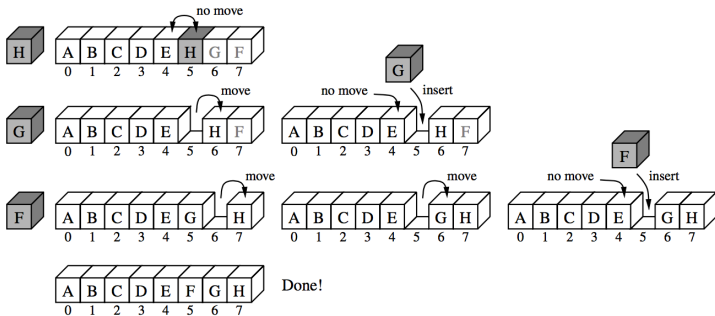
## 算法描述

- 首先考虑数组中的第一个元素。注意一个元素本身已经排序。
- 再考虑下一个元素。如果它比第一个小，则交换它们。
- 接下来考虑第三个元素，将其向左交换，直到与前两个元素的顺序正确为止。
- 然后考虑第四个元素，将其向左交换，直到与前三个元素的顺序正确为止。
- 以这种方式继续使用第五个元素、第六个元素等，直到对整个数组进行排序。

# Insert-Sorting



# Insert-Sorting





# Insert-Sorting

```
def insertion_sort(A):  
    for k in range(1, len(A)):  
        cur = A[k]  
        j = k  
        while j > 0 and A[j-1] > cur:  
            A[j] = A[j-1]  
            j -= 1  
        A[j] = cur
```

# Insert-Sorting

```
if __name__ == '__main__':  
    A = [5, 6, 4, 3]  
    print('before sort, A = ', A)  
    insertion_sort(A)  
    print('after sort, A = ', A)
```

# Insert-Sorting

```
if __name__ == '__main__':  
    A = [5, 6, 4, 3]  
    print('before sort, A = ', A)  
    insertion_sort(A)  
    print('after sort, A = ', A)
```

```
before sort, A =  [5, 6, 4, 3]  
after sort, A =   [3, 4, 5, 6]
```