

云南大学数学与统计学院
上机实践报告

课程名称：数据结构与算法实验	年级：2015 级	上机实践成绩：
指导教师：陆正福	姓名：刘鹏	
上机实践名称：数组序列实验	学号：20151910042	上机实践日期：2017-06-08
上机实践编号：No.05	组号：	上机实践时间：上午 3、4 节

一、实验目的

1. 熟悉与 Python 序列类型有关的数据结构与算法；
2. 熟悉主讲教材 Chapter 5 的代码片段。

二、实验内容

1. 数组序列有关的数据结构设计 with 算法设计
2. 调试主讲教材 Chapter 5 的 5.5.3 节的 Simple Cryptography 程序

三、实验平台

Windows 10 1703 Enterprise 中文版；
Python 3.6.0；
Wing IDE Professional 6.0.5-1 集成开发环境；
MATLAB R2017a；
Microsoft Office Excel 2016.

四、实验记录与实验结果分析

1 题

动态数组与均摊方法

程序代码：

```
1 # 5.3.0 Dynamic Arrays and Amortization
2
3 import sys                # provides getsizeof function
4 data = []
5 for k in range(20):       # NOTE: must fix choice of n
6     a = len(data)         # number of elements
7     b = sys.getsizeof(data) # actual size in bytes
8     print('Length:{0:3d}; Size in bytes:{1:4d}'.format(a,b))
9     data.append(None)     # increase by one
```

程序代码 1

实验结果：

5.3.0 SIMPLE TEST.py	Debug process terminated
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
Length: 6;	Size in bytes: 128
Length: 7;	Size in bytes: 128
Length: 8;	Size in bytes: 128
Length: 9;	Size in bytes: 192
Length: 10;	Size in bytes: 192
Length: 11;	Size in bytes: 192
Length: 12;	Size in bytes: 192
Length: 13;	Size in bytes: 192
Length: 14;	Size in bytes: 192
Length: 15;	Size in bytes: 192
Length: 16;	Size in bytes: 192
Length: 17;	Size in bytes: 264
Length: 18;	Size in bytes: 264
Length: 19;	Size in bytes: 264

运行结果 1

代码分析：

这是个分析数组大小与内存占用量的函数，通过与系统交互，得到实际数据。很显然，理论数据与内存占用不相同，正式因为 Python 是动态语言。这里的实验仅仅是一个尝试，课本后面有这后面有着详细的解释。这是一种“等比性质”的动态数组，有着很多优越性。

2 题

动态数组的实现。

程序代码：

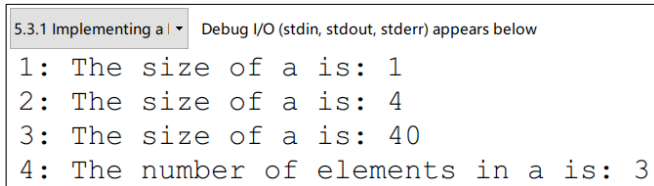
```

1  # 5.3.1 Implementing a Dynamic Array
2
3  import ctypes                # provides low-level arrays
4
5  class DynamicArray:
6      """A dynamic array class akin to a simplified Python list."""
7
8      def __init__(self):
9          """Create an empty array."""
10         self._n = 0            # count actual elements
11         self._capacity = 1    # default array capacity
12         self._A = self._make_array(self._capacity) # low-level array
13
14     def __len__(self):
15         """Return number of elements stored in the array."""
16         return self._n
17
18     def __getitem__(self, k):
19         """Return element at index k."""
20         if not 0 <= k < self._n:
21             raise IndexError('invalid index')
22         return self._A[k]      # retrieve from array
23
24     def append(self, obj):
25         """Add object to end of the array."""
26         if self._n == self._capacity: # not enough room
27             self._resize(2 * self._capacity) # so double capacity
28         self._A[self._n] = obj
29         self._n += 1
30
31     def _resize(self, c): # nonpublic utility
32         """Resize internal array to capacity c."""
33         B = self._make_array(c) # new (bigger) array
34         for k in range(self._n): # for each existing value
35             B[k] = self._A[k]
36         self._A = B # use the bigger array
37         self._capacity = c
38
39     def _make_array(self, c): # nonpublic utility
40         """Return new array with capacity c."""
41         return (c * ctypes.py_object)() # see ctypes documentation
42
43 #----- my main function -----
44
45 a = DynamicArray()
```

```
46 print('1: The size of a is:',a._capacity)
47 a.append(1)
48 a.append(2)
49 a.append(3)
50 print('2: The size of a is:',a._capacity)
51 a._resize(40)
52 print('3: The size of a is:',a._capacity)
53 print('4: The number of elements in a is:',a._n)
```

程序代码 2

运行结果:



```
5.3.1 Implementing a | ▾ Debug I/O (stdin, stdout, stderr) appears below
1: The size of a is: 1
2: The size of a is: 4
3: The size of a is: 40
4: The number of elements in a is: 3
```

运行结果 2

代码分析:

这个代码仅仅作为一个参考,但是实际并不能用,因为 `DynamicArray` 类的封装做得很差,函数是用单下划线开头的保护类型,这里为了安全起见,应该将之设计为 `private` 类型,即双下划线开头。这样一来,类的使用者就看不到 `capacity` 和 `n`,更无法修改这两个数值,这就安全了很多。

值得指出的就是 `_make_array` 函数里的 `ctypes.py_object` 函数了,这里用 `help(ctypes.py_object)` 获取了帮助文档。

关于 ctypes 的一些方法，doc 如下：

```

1  Help on class py_object in module ctypes:
2
3  class py_object(_ctypes._SimpleCData)
4  |   XXX to be provided
5
6  |   Method resolution order:
7  |       py_object
8  |       _ctypes._SimpleCData
9  |       _ctypes._CData
10 |       builtins.object
11
12 |   Methods defined here:
13
14 |   __repr__(self)
15 |       Return repr(self).
16
17 |   -----
18 |   Data descriptors defined here:
19
20 |   __dict__
21 |       dictionary for instance variables (if defined)
22
23 |   __weakref__
24 |       list of weak references to the object (if defined)
25
26 |   -----
27 |   Methods inherited from _ctypes._SimpleCData:
28
29 |   __bool__(self, /)
30 |       self != 0
31
32 |   __ctypes_from_outparam__(...)
33
34 |   __init__(self, /, *args, **kwargs)
35 |       Initialize self.  See help(type(self)) for accurate signature.
36
37 |   __new__(*args, **kwargs) from _ctypes.PyCSimpleType
38 |       Create and return a new object.  See help(type) for accurate signature.
39
40 |   -----
41 |   Data descriptors inherited from _ctypes._SimpleCData:
42
43 |   value
44 |       current value
45
46 |   -----
47 |   Methods inherited from _ctypes._CData:
48
49 |   __hash__(self, /)
50 |       Return hash(self).
51
52 |   __reduce__(...)
53 |       helper for pickle
54
55 |   __setstate__(...)

```

3 题

实验 Python 动态数组的均摊时间复杂度。

程序代码：

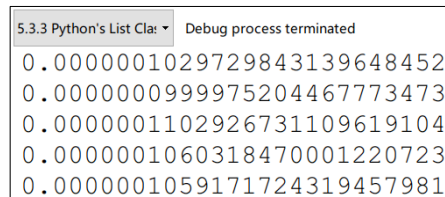
```

1  # 5.3.3 Python's List Class
2
3  from time import time      # import time function from time module
4  def compute_average(n):
5      """Perform n appends to an empty list and return average time elapsed."""
6      data = []
7      start = time()         # record the start time (in seconds)
8      for k in range(n):
9          data.append(None)
10     end = time()            # record the end time (in seconds)
11     return (end - start)/n # compute average per operation
12
13  #----- my main function -----
14
15  a = []
16  n = 10000
17  while n <= 100000000:
18      print(format(compute_average(n), '1.25f'))
19      n *= 10

```

程序代码 3

运行结果：



```

5.3.3 Python's List Cla... Debug process terminated
0.0000001029729843139648452
0.0000000999975204467773473
0.0000001102926731109619104
0.0000001060318470001220723
0.0000001059171724319457981

```

运行结果 3

代码分析：

可以从上面的数据中看到，平均 `append` 一个元素的耗费时间从平均上看是一样的。所以，这种扩充容量的方法是有效的。即保持在扩充数组的时候，数组容量与所扩充的容量的比例。

值得指出的是，`resize` 是动态的，这点至关重要。必须要保证指数增长，才能使得均摊之后的量级是线性的。固定增长，会带来平方量级的 `append`。

4 题

DynamicArray 的 insert 与 remove 操作实验。

程序代码:

```

1  # 5.4.1 Efficiency of Python's List and Tuple Classes
2
3  import ctypes
4
5  class DynamicArray:
6      """A dynamic array class akin to a simplified Python list."""
7
8      def __init__(self):
9          """Create an empty array."""
10         self._n = 0          # count actual elements
11         self._capacity = 1   # default array capacity
12         self._A = self._make_array(self._capacity) # low-level array
13
14     def __len__(self):
15         """Return number of elements stored in the array."""
16         return self._n
17
18     def __getitem__(self, k):
19         """Return element at index k."""
20         if not 0 <= k < self._n:
21             raise IndexError('invalid index')
22         return self._A[k]      # retrieve from array
23
24     def append(self, obj):
25         """Add object to end of the array."""
26         if self._n == self._capacity:      # not enough room
27             self._resize(2 * self._capacity) #so double capacity
28         self._A[self._n] = obj
29         self._n += 1
30
31     def _resize(self, c):                  # nonpublic utility
32         """Resize internal array to capacity c."""
33         B = self._make_array(c)           # new (bigger) array
34         for k in range(self._n):          # for each existing value
35             B[k] = self._A[k]
36         self._A = B                       # use the bigger array
37         self._capacity = c
38
39     def _make_array(self, c):              # nonpublic utility
40         """Return new array with capacity c."""
41         return(c * ctypes.py_object)()    # see ctypes documentation
42
43     def insert(self, k, value):
44         """Insert value at index k, shifting subsequent values rightward."""
45         # (for simplicity, we assume 0 <= k <= n in this version)

```

```

46         if self._n == self._capacity:           # not enough room
47             self._resize(2 * self._capacity)     # so double capacity
48         for j in range(self._n, k, -1):           # shift rightmost first
49             self._A[j] = self._A[j-1]
50         self._A[k] = value                         # store newest element
51         self._n += 1
52
53     def remove(self, value):
54         """Remove first occurrence of value (or raise ValueError)."""
55         # note: we do not consider shrinking the dynamic array in this version
56         for k in range(self._n):
57             if self._A[k] == value:               # found a match
58                 for j in range(k, self._n - 1):   # shift others to fill gap
59                     self._A[j] = self._A[j+1]
60                 self._A[self._n - 1] = None        # help garbage collection
61                 self._n -= 1                       # we have one less it
62                 return                             # exit immediately
63         raise ValueError('value not found')       # only reached if no match
64
65     #----- my main function -----
66
67     from time import time
68     N = 100
69     Time_Head = []
70     Time_Middle = []
71     Time_Tail = []
72     while N <= 1000000:
73         L = DynamicArray()
74         for i in range(N):
75             L.append(None)
76         begin = time()
77         L.insert(0, None)
78         end = time()
79         Time_Head.append(end - begin)
80         N *= 10
81
82     N = 100
83     while N <= 1000000:
84         L = DynamicArray()
85         for i in range(N):
86             L.append(None)
87         begin = time()
88         L.insert(N // 2, None)
89         end = time()
90         Time_Middle.append(end - begin)
91         N *= 10
92
93     N = 100
94     while N <= 1000000:

```



```
95     L = DynamicArray()
96     for i in range(N):
97         L.append(None)
98     begin = time()
99     L.insert(N, None)
100    end = time()
101    Time_Tail.append(end - begin)
102    N *= 10
103
104    print('----- Add Head -----')
105    for i in range(5):
106        print(format(Time_Head[i], '1.25f'))
107
108    print('----- Add Middle -----')
109    for i in range(5):
110        print(format(Time_Middle[i], '1.25f'))
111
112    print('----- Add Tail -----')
113    for i in range(5):
114        print(format(Time_Tail[i], '1.25f'))
115
116    print('----- RM Head -----')
117
118    for i in range(20):
119        L = DynamicArray()
120        for j in range(10000):
121            L.append(j)
122        begin = time()
123        L.remove(L[i * 500])
124        end = time()
125        print(format(end - begin, '1.25f'))
```

程序代码 4

运行结果：

```
5.4.1 Efficiency of Pyth ▾ Debug process terminated
----- Add Head -----
0.00000000000000000000000000000000
0.00000000000000000000000000000000
0.0020043849945068359375000
0.0180480480194091796875000
0.1915094852447509765625000
----- Add Middle -----
0.0005013942718505859375000
0.00000000000000000000000000000000
0.0010027885437011718750000
0.0090243816375732421875000
0.0897686481475830078125000
----- Add Tail -----
0.00000000000000000000000000000000
0.00000000000000000000000000000000
0.00000000000000000000000000000000
0.00000000000000000000000000000000
0.00000000000000000000000000000000
----- RM Head -----
0.0029823780059814453125000
0.0034825801849365234375000
0.0030078887939453125000000
0.0030081272125244140625000
0.0030078887939453125000000
0.0030078887939453125000000
0.0030081272125244140625000
0.0025064945220947265625000
0.0025067329406738281250000
0.0025064945220947265625000
0.0020053386688232421875000
0.0025064945220947265625000
0.0020053386688232421875000
0.0020051002502441406250000
0.0020055770874023437500000
0.0015039443969726562500000
0.0015039443969726562500000
0.0015039443969726562500000
0.0010027885437011718750000
0.0010030269622802734375000
```

运行结果 4

代码分析：

由于这里添加了两个方法，所以需要将之前的代码抄过来。暂时不使用继承了，那样会更加麻烦。

可以看到，在 insert 操作中，对于相同量级的数组，在越靠后的位置插入，花费越少，反之相反。而对于 remove 操作，由于代码还是不够好，没有给出缩小容量的 resize 操作，所以这里就简单看看。从下图可以看到，运行时间与移除位置有着负相关性。越靠后，越好移动，而且相关性基本是线性的。

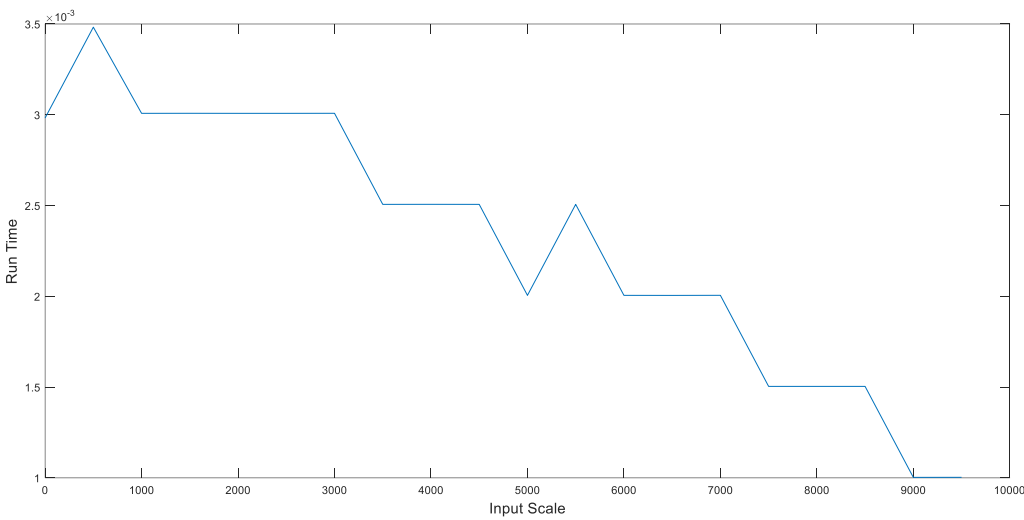


Figure 1

5 题

字符串列表的效率分析。

程序代码：

```

1  # 5.4.2 Python's String Class
2
3  import random
4  from time import time
5
6  Random = 'abcdefghijklmnopqrstuvwxyz123456789!@#$%^&*()_-=[]{}|\\/?<>,.`~` '
7  document = str()
8  document = ''.join(random.choice(Random) for i in range(1000000))
9
10 # first time
11 letters = ''
12 begin = time()
13 for c in document:
14     if c.isalpha():
15         letters += c
16 end = time()
17 print('1:',end - begin)
18
19 # second time
20 letters = ''
21 temp = []
22 begin = time()
23 for c in document:
24     if c.isalpha():
25         temp.append(c)
26 letters = ''.join(temp)
27 end = time()
28 print('2:',end - begin)
29
30 # third time
31 letters = ''
32 begin = time()
33 letters = ''.join(c for c in document if c.isalpha())
34 end = time()
35 print('3:',end - begin)
36
37 # forth time
38 letters = ''
39 begin = time()
40 letters = ''.join([c for c in document if c.isalpha()])
41 end = time()
42 print('4:',end - begin)

```

程序代码 5

运行结果:

```
5.4.2 Python's String C Debug I/O (stdin, stdout, stderr) appears below
1: 0.17499494552612305
2: 0.1669473648071289
3: 0.26220035552978516
4: 0.07319450378417969
```

代码分析:

代码片段，自己动手写了几行程序，测试一下时间。可以发现，第四种，即方括号类型的单行表达式最节省时间。

在生成 `document` 文档的时候，采用了选择性的 `random` 方法。设计一个目标序列，然后通过这个序列生成随机 `document`，之后就可以统计与时间计数。

6 题

高分榜程序的实现。

程序代码:

```

1  # 5.5.1 Storing High Scores for a Game
2
3  class GameEntry:
4      """Represents one entry of a list of high scores."""
5
6      def __init__(self,name,score):
7          self._name = name
8          self._score = score
9
10         def get_name(self):
11             return self._name
12         def get_score(self):
13             return self._score
14         def __str__(self):
15             return '({0}, {1})'.format(self._name,self._score) # e.g., 'Bob,98'
16
17 class Scoreboard():
18     """Fixed-length sequence of high scores in nondecreasing order."""
19
20     def __init__(self,capacity = 10):
21         """Initialize scoreboard with given maximum capacity.
22
23         All entries are initially None.
24         """
25         self._board = [None] * capacity # reserve space for future scores
26         self._n = 0 # number of actual entries
27
28     def __getitem__(self,k):
29         """Return entry at index k."""
30         return self._board[k]
31
32     def __str__(self):
33         """Return string representation of the high score list."""
34         return '\n'.join(str(self._board[j]) for j in range(self._n))
35
36     def add(self,entry):
37         """Consider adding entry to high scores."""
38         score = entry.get_score()
39
40         # Does new entry qualify as a high score?
41         # answer is yes if board not full or score is higher than last entry
42         good = self._n < len(self._board) or score > self._board[-1].get_score()
43
44         if good:
45             if self._n < len(self._board): # no score drops from list
46                 self._n += 1 # so overall number increases

```

```

46
47     # shift lower scores rightward to make room for new entry
48     j = self._n - 1
49     while j > 0 and self._board[j-1].get_score() < score:
50         self._board[j] = self._board[j-1]    # shift entry from j-1 to j
51         j -= 1                                # and decrement j
52     self._board[j] = entry                    # when done, add new entry
53
54 #----- my main function -----
55
56 a = Scoreboard(3)
57
58 s_1 = GameEntry('LiuPeng',93)
59 s_2 = GameEntry('zhangDi',98)
60 s_3 = GameEntry('LiYue',22)
61
62 a.add(s_1)
63 a.add(s_2)
64 a.add(s_3)
65 print('----- first record -----')
66 print(a.__str__())
67 print('----- get item -----')
68 print(a.__getitem__(2))
69 s_4 = GameEntry('Torvalds',100)
70 a.add(s_4)
71 print('----- second record -----')
72 print(a.__str__())

```

程序代码 6

运行结果:

```

5.5.1 Storing High Sc... Debug I/O (stdin, stdout, stderr) appears below
----- first record -----
(zhangDi, 98)
(LiuPeng, 93)
(LiYue, 22)
----- get item -----
(LiYue, 22)
----- second record -----
(Torvalds, 100)
(zhangDi, 98)
(LiuPeng, 93)

```

代码分析:

这段代码相对比较简单, 核心就是两个类的同时调用。GameEntry 的实例作为 Scoreboard 类的初始化成员。

除了用两个平行类进行调用之外, 还可以对数据进行封装, 把两个类写成一个, 即把 GameEntry 作为 Nested class 封装进 Scoreboard 里面, 进行安全处理。代码如下。

程序代码改进:

```

1  # 5.5.1 Storing High Scores for a Game_Optimistic version
2
3  class Scoreboard():
4      """Fixed-length sequence of high scores in nondecreasing order."""
5
6      class _GameEntry:
7          """Nonpublic class for storing entry.
8
9          Represents one entry of a list of high scores."""
10         __slots__ = '_name', '_score'
11         def __init__(self, name, score):
12             self._name = name
13             self._score = score
14         def get_name(self):
15             return self._name
16         def get_score(self):
17             return self._score
18         def __str__(self):
19             return '({0}, {1})'.format(self._name, self._score) # e.g., 'Bob, 98'
20     def __init__(self, capacity = 10):
21         """Initialize scoreboard with given maximum capacity.
22
23         All entries are initially None.
24         """
25         self._board = [None] * capacity # reserve space for future scores
26         self._n = 0 # number of actual entries
27
28     def __getitem__(self, k):
29         """Return entry at index k."""
30         return self._board[k]
31
32     def __str__(self):
33         """Return string representation of the high score list."""
34         return '\n'.join(str(self._board[j]) for j in range(self._n))
35
36     def add(self, name, score):
37         """Consider adding entry to high scores."""
38         entry = self._GameEntry(name, score)
39         score = entry.get_score()
40         # Does new entry qualify as a high score?
41         # answer is yes if board not full or score is higher than last entry
42         good = self._n < len(self._board) or score > self._board[-1].get_score()
43
44         if good:
45             if self._n < len(self._board): # no score drops from list
46                 self._n += 1 # so overall number increases
47
48             # shift lower scores rightward to make room for new entry

```



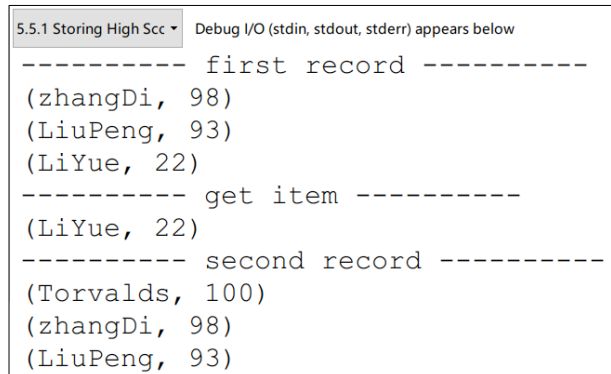
```

49         j = self._n - 1
50         while j > 0 and self._board[j-1].get_score() < score:
51             self._board[j] = self._board[j-1]    # shift entry from j-1 to j
52             j -= 1                                # and decrement j
53             self._board[j] = entry                # when done, add new entry
54
55     #----- my main function -----
56
57     a = Scoreboard(3)
58     a.add('LiuPeng',93)
59     a.add('zhangDi',98)
60     a.add('LiYue',22)
61     print('----- first record -----')
62     print(a.__str__())
63     print('----- get item -----')
64     print(a.__getitem__(2))
65
66     a.add('Torvalds',100)
67     print('----- second record -----')
68     print(a.__str__())

```

程序代码 7

运行结果:



```

5.5.1 Storing High Sc... Debug I/O (stdin, stdout, stderr) appears below
----- first record -----
(zhangDi, 98)
(LiuPeng, 93)
(LiYue, 22)
----- get item -----
(LiYue, 22)
----- second record -----
(Torvalds, 100)
(zhangDi, 98)
(LiuPeng, 93)

```

运行结果 5

代码分析:

这段优化的代码，即把节点类进行封装，借此实现数据的保护。可以看到，`GameEntry` 类的前边加了下划线，表示是被保护的。

7 题

选择排序。

程序代码：

```
1 # 5.5.2 Sorting a sequence
2
3 def insertion_sort(A):
4     """Sort list of comparable elements into nondecreasing order."""
5     for k in range(1, len(A)):          # from 1 to n-1
6         cur = A[k]                      # current element to be inserted
7         j = k                            # find correct index j for current
8         while j > 0 and A[j-1] > cur:    # element A[j-1] must be after current
9             A[j] = A[j-1]
10            j -= 1
11        A[j] = cur                       # cur is now in the right place
12
13 #----- my main function -----
14
15 import random
16 a = [random.randint(0,100) for i in range(10)]
17 print('1:', a)
18 insertion_sort(a)
19 print('2:', a)
```

程序代码 8

运行结果：

```
5.5.2 Sorting a sequer  Debug I/O (stdin, stdout, stderr) appears below
1: [90, 66, 56, 51, 19, 96, 54, 44, 94, 2]
2: [2, 19, 44, 51, 54, 56, 66, 90, 94, 96]
```

运行结果 6

代码分析：

插入排序很简单，效率也很低。时间复杂度是平方量级。

这个算法可以在后期的排序与选择章节中详细谈论，这里的理解，可以跟玩扑克牌做对比，摸牌插入，基本就是这个原理。

8 题

简单的凯撒加密。

程序代码：

```

1  # 5.5.3 Simple Cryptography
2
3  class CaesarCipher:
4      """Class for doing encryption and decryption using a Caesar cipher."""
5
6      def __init__(self, shift):
7          """Construct Caesar cipher using given integer shift for rotation."""
8          encoder = [None] * 26          # temp array for encryption
9          decoder = [None] * 26          # temp array for decryption
10         for k in range(26):
11             encoder[k] = chr((k + shift) % 26 + ord('A'))
12             decoder[k] = chr((k - shift) % 26 + ord('A'))
13             # shift is the step you set for the code
14         self._forward = ''.join(encoder)  # will store as string
15         self._backward = ''.join(decoder) # since fixed
16
17     def encrypt(self, message):
18         """Return string representing encrypted message."""
19         return self._transform(message, self._forward)
20
21     def decrypt(self, secret):
22         """Return decrypted message given encrypted secret."""
23         return self._transform(secret, self._backward)
24
25     def _transform(self, original, code):
26         """Utility to perform transformation based on given code string."""
27         msg = list(original)
28         for k in range(len(msg)):
29             if msg[k].isupper():
30                 j = ord(msg[k]) - ord('A') # index from 0 to 25
31                 msg[k] = code[j]           # replace this character
32         return ''.join(msg)
33
34     #----- book's function -----
35
36     if __name__ == '__main__':
37         cipher = CaesarCipher(3)
38         message = "THE EAGLE IS IN PLAY; MEET AT JOE'S."
39         coded = cipher.encrypt(message)
40         print('Secret: ', coded)
41         answer = cipher.decrypt(coded)
42         print('Message: ', answer)

```

程序代码 9

运行结果：

5.5.3 Simple Cryptogr ▾ Debug process terminated

Secret: WKH HDJOH LV LQ SODB; PHHW DW MRH'V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S.

运行结果 7

代码分析：

9 题

井字棋计分板。

这里有必要说一下 str.join 函数的作用

```
1 In [0]: str.join?
2 Docstring:
3 S.join(iterable) -> str
4
5 Return a string which is the concatenation of the strings in the
6 iterable. The separator between elements is S.
7 Type:      method_descriptor
```

程序代码:

```
1 # 5.6.0 Multidimensional Data Sets
2
3 class TicTacToe:
4     """Management of a Tic-Tac-Toe game (does not do strategy)."""
5
6     def __init__(self):
7         """Start a new game."""
8         self._board = [[' ']*3 for j in range(3)]
9         self._player = 'X'
10
11    def mark(self,i,j):
12        """Put an X or O mark at position (i,j) for next player's turn."""
13        if not (0 <= i <= 2 and 0 <= j <= 2):
14            raise ValueError('Invalid board position')
15        if self._board[i][j] != ' ':
16            raise ValueError('Board position occupied')
17        if self.winner() is not None:
18            raise ValueError('Game is already complete')
19        self._board[i][j] = self._player
20        if self._player == 'X':
21            self._player = 'O'
22        else:
23            self._player = 'X'
24
25    def _is_win(self,mark):
26        """Check whether the board configuration is a win for the given player."""
27        board = self._board # local variable for shorthand
28        return (mark == board[0][0] == board[0][1] == board[0][2] or # row 0
29                mark == board[1][0] == board[1][1] == board[1][2] or # row 1
30                mark == board[2][0] == board[2][1] == board[2][2] or # row 2
31                mark == board[0][0] == board[1][0] == board[2][0] or # column 0
32                mark == board[0][1] == board[1][1] == board[2][1] or # column 1
33                mark == board[0][2] == board[1][2] == board[2][2] or # column 2
34                mark == board[0][0] == board[1][1] == board[2][2] or # diagonal
35                mark == board[0][2] == board[1][1] == board[2][0]) # rev diag
36
```

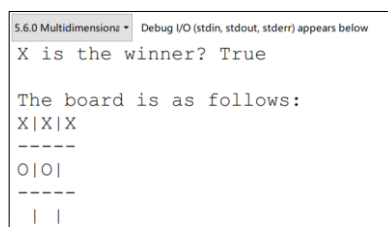
```

37     def winner(self):
38         """Return mark of winning player, or None to indicate a tie."""
39         for mark in 'XO':
40             if self._is_win(mark):
41                 return mark
42         return None
43
44     def __str__(self):
45         """Return string representation of current game board."""
46         rows = ['|'.join(self._board[r]) for r in range(3)]
47         return '\n-----\n'.join(rows)
48
49 #----- my main function -----
50
51 B = TicTacToe()
52 B.mark(0,0)
53 B.mark(1,0)
54 B.mark(0,1)
55 B.mark(1,1)
56 B.mark(0,2)
57 print('X is the winner?',B._is_win('X'))
58 print('\nThe board is as follows:')
59 print(B.__str__())

```

程序代码 10

运行结果:



```

5.6.0 Multidimensional array - Debug I/O (stdin, stdout, stderr) appears below
X is the winner? True

The board is as follows:
X|X|X
-----
O|O|
-----
| |

```

运行结果 8

代码分析:

这是个简单的记分牌程序，可以记录游戏的状态，用的结构是二维数组，或者说 list of lists。没有难度。

值得注意的是，程序最后的输出是用字符串类型的图形输出棋盘。很不错。值得学习。

五、教材翻译

Translation

Chapter 5 Array-Based Sequence

* 第五章 基于数组的序列

5.1 Python's Sequence Types

* 5.1 节 Python 的序列类型

In this chapter, we explore Python's various "sequence" classes, namely the built-in **list**, **tuple**, and **str** classes. There is significant commonality between these classes, most notably: each supports indexing to access an individual element of a sequence, using a syntax such as `seq[k]`, and each uses a low-level concept known as an **array** to represent the sequence. However, there are significant differences in the abstractions that these classes represent, and in the way that instances of these classes are represented internally by Python. Because these classes are used so widely in Python programs, and because they will become building blocks upon which we will develop more complex data structures, it is imperative that we establish a clear understanding of both the public behavior and inner workings of these classes.

* 在这一章中，我们将要探究一下 Python 语言中多变的序列类，也就是列表、元组还有字符串这三种内建类。这三种类之间有着很明显的共性，最明显的就是都支持通过下标来访问序列中的每一个元素，这可以用 `seq[k]` 这个语句来实现。并且这三个类都是通过一种叫做数组的底层概念来进行表达的。然而，在这些类的抽象表达中仍然有着明显的差异，而且正是通过这些差异 Python 才在语言内部能支持这些类的实例。我们之所以如此想要将这些类的公共行为与内部工作机制研究透彻，既是因为这些类在 Python 程序中被广泛使用，又是因为这些类能构建许多复杂的数据结构。

Public Behaviors

* 公共行为

A proper understanding of the outward semantics for a class is a necessity for a good programmer. While the basic usage of lists, strings, and tuples may seem straightforward, there are several important subtleties regarding the behaviors associated with these classes (such as what it means to make a copy of a sequence, or to take a slice of a sequence). Having a misunderstanding of a behavior can easily lead to inadvertent bugs in a program. Therefore, we establish an accurate

mental model for each of these classes. These images will help when exploring more advanced usage, such as representing a multidimensional data set as a list of lists.

* 正确理解一个类是成为一个好的程序员的必要条件。虽然列表，字符串和元组的基本用法可能看起来很简单，但是与这些类相关联的行为却有几个重要的细节差别（例如，生成序列的副本或取一个序列的片段）。对行为的误解容易导致程序中的无意的错误。因此，我们为每一个类都建立了准确的模型。这些模型将有助于探索更高级的用法，例如将多维数据集表示为列表。

Implementation Details

* 实现细节

A focus on the internal implementations of these classes seems to go against our stated principles of object-oriented programming. In Section 2.1.2, we emphasized the principle of **encapsulation**, noting that the user of a class need not know about the internal details of the implementation. While it is true that one only needs to understand the syntax and semantics of a class's public interface in order to be able to write legal and correct code that uses instances of the class, the efficiency of a program depends greatly on the efficiency of the components upon which it relies.

* 关注这些类的内部实现细节似乎违背了我们所陈述的面向对象编程原理。在 2.1.2 节中，我们强调了封装的原理，注意到一个类的用户不需要了解类的内部细节。虽然只需要了解类的公共接口的语法和语义，就可以编写调用类的实例以及写出合法和正确的代码，但是程序的执行效率还是在很大程度上取决于构成程序的组件。

Asymptotic and Experimental Analyses

* 渐近和实验分析

In describing the efficiency of various operations for Python's sequence classes, we will rely on the formal **asymptotic analysis** notations established in Chapter 3. We will also perform experimental analyses of the primary operations to provide empirical evidence that is consistent with the more theoretical asymptotic analyses.

* 在描述 Python 序列类的各种操作的效率时，我们将依靠第 3 章中建立的形式渐近分析符号。我们还将对主要操作进行实验分析，提供与理论渐近分析一致的经验证据。

5.2 Low-Level Array

* 底层数组

To accurately describe the way in which Python represents the sequence types, we must first discuss aspects of the low-level computer architecture. The primary memory of a computer is composed of bits of information, and those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a *byte*, which is equivalent to 8 bits.

* 为了准确地描述 Python 表示序列类型的方式，我们首先讨论低级别计算机体系结构的各个方面。计算机的主存储器由信息比特组成，并且这些比特通常被分组成较大单位，而这些较大的单位是可以由系统架构精确控制的。这样一个典型的较大的单位是字节，相当于 8 比特。

A computer system will have a huge number of bytes of memory, and to keep track of what information is stored in what byte, the computer uses an abstraction known as a *memory address*. In effect, each byte of memory is associated with a unique number that serves as its address (more formally, the binary representation of the number serves as the address). In this way, the computer system can refer to the data in “byte #2150” versus the data in “byte #2157,” for example. Memory addresses are typically coordinated with the physical layout of the memory system, and so we often portray the numbers in sequential fashion. Figure 5.1 provides such a diagram, with the designated memory address for each byte.

* 计算机系统具有大容量的存储器，可以存放很多的字节，并且为了跟踪信息到底被存储在哪个字节中，计算机使用了称为存储器地址的一种抽象概念。实际上，存储器的一个字节与用作其地址的一个数字（二进制）相关联。以这种方式，计算机系统可以参考“字节 # 2150”中的数据与“字节 # 2157”中的数据。存储器地址通常与存储器系统的物理布局协调，因此我们经常以连续的方式描绘数字。图 5.1 提供了这样一个图，每个字节都有指定的存储器地址。



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

Despite the sequential nature of the numbering system, computer hardware is designed, in theory, so that any byte of the main memory can be efficiently accessed based upon its memory address. In this sense, we say that a computer's main memory performs as *random access memory (RAM)*. That is, it is just as easy to retrieve byte #8675309 as it is to retrieve byte #309. (In practice, there are complicating factors including the use of caches and external memory; we address some of those issues in Chapter 15.) Using the notation for asymptotic analysis, we say that any individual byte of memory can be stored or retrieved in $O(1)$ time.

* 编号系统具有连续的性质，所以理论上设计的计算机硬件，可以基于其存储器地址有效地访问主存储器的任

何字节。在这个意义上说，一台计算机的主存储器是随机存取存储器（RAM）。也就是说，检索字节#8675309就像检索字节#309一样容易。（实际上，存在复杂因素，包括使用缓存和外部存储器；我们将在第 15 章中的解决这里的部分问题。）使用渐近分析的方式，可以将任何单个字节内存存储或检索的时间复杂度降低至 $O(1)$ 。

In general, a programming language keeps track of the association between an identifier and the memory address in which the associated value is stored. For example, identifier *x* might be associated with one value stored in memory, while *y* is associated with another value stored in memory. A common programming task is to keep track of a sequence of related objects. For example, we may want a video game to keep track of the top ten scores for that game. Rather than use ten different variables for this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group.

* 通常，编程语言时刻保持着标识符和存储器地址之间的联系。例如，标识符 *x* 可能与存储在存储器中的一个值相关联，而 *y* 与存储在存储器中的另一个值相关联。常见的编程任务是跟踪一系列相关对象。例如，我们可能想要一个电脑游戏跟踪该游戏的前十名，我们更倾向于使用一个数组进行下标索引，而不是设置十个不同的变量。

A group of related variables can be stored one after another in a contiguous portion of the computer's memory. We will denote such a representation as an *array*. As a tangible example, a text string is stored as an ordered sequence of individual characters. In Python, each character is represented using the Unicode character set, and on most computing systems, Python internally represents each Unicode character with 16 bits (i.e., 2 bytes). Therefore, a six-character string, such as *SAMPLE*, would be stored in 12 consecutive bytes of memory, as diagrammed in Figure 5.2.

* 一组相关变量可以一个接一个地存储在计算机内存的连续部分。我们将这样的表示形式命名为数组。比如，文本字符串被存储为单个字符的有序序列。在 Python 中，每个字符都使用 Unicode 字符集来表示，而在大多数计算系统下，Python 内部用 16 位（即 2 字节）表示一个 Unicode 字符。因此，如图 5.2 所示，六个字符的字符串（如 *SAMPLE*）将被存储在 12 个连续字节的存储器中。

We describe this as an *array of six characters*, even though it requires 12 bytes of memory. We will refer to each location within an array as a *cell*, and will use an integer *index* to describe its location within the array, with cells numbered starting with 0, 1, 2, and so on. For example, in Figure 5.2, the cell of the array with index 4 has contents *L* and is stored in bytes 2154 and 2155 of memory.

* 我们将其描述为六个字符的数组，虽然它需要 12 个字节的内存。我们把数组中的每个位置称为单元格，并使用整数索引来描述其在数组中的位置，单元格以 0, 1,

2 等开始。例如，在图 5.2 中，具有索引 4 的数组的单元格存储有字符 L，并存储在存储器的 2154 和 2155 字节中。

Each cell of an array must use the same number of bytes. This requirement is what allows an arbitrary cell of the array to be accessed in constant time based on its index. In particular, if one knows the memory address at which an array starts (e.g., 2146 in Figure 5.2), the number of bytes per element (e.g., 2 for a Unicode character), and a desired index within the array, the appropriate memory address can be computed using the calculation, $\text{start} + \text{cellsize} * \text{index}$. By this formula, the cell at index 0 begins precisely at the start of the array, the cell at index 1 begins precisely cellsize bytes beyond the start of the array, and so on. As an example, cell 4 of Figure 5.2 begins at memory location $2146 + 2 * 4 = 2146 + 8 = 2154$.

* 数组的每个单元必须使用相同的字节数。这个为了实现通过索引访问任意单元而确定的。特别地，如果知道

数组开始的存储器地址（例如，图 5.2 中的 2146）以及每个元素的字节数（例如，Unicode 字符的 2 个字节）和阵列内的所需索引，可以使用 $\text{start} + \text{cellsize} * \text{index}$ 计算单元的地址。通过该公式，下标 0 处的单元格位于数组的开头，下标 1 处的单元格从下标 0 元素之后 cellsize 个字节处开始，等等。作为示例，图 5.2 的单元 4 从存储器位置 $2146 + 2 * 4 = 2146 + 8 = 2154$ 开始。

Of course, the arithmetic for calculating memory addresses within an array can be handled automatically. Therefore, a programmer can envision a more typical high-level abstraction of an array of characters as diagrammed in Figure 5.3.

* 当然，可以自动计算数组内元素的存储地址。因此，程序员可以构建一个更典型的高级抽象字符数组，如图 5.3 所示。

5.2.1 Referential Arrays

* 参考数组

As another motivating example, assume that we want a medical information system to keep track of the patients currently assigned to beds in a certain hospital. If we assume that the hospital has 200 beds, and conveniently that those beds are numbered from 0 to 199, we might consider using an array-based structure to maintain the names of the patients currently assigned to those beds. For example, in Python we might use a list of names, such as:

* 举例说来, 假设我们想要开发一个医疗信息系统, 用以跟踪被分配到某个医院的患者。如果假设医院有 200 张病床, 这些病床的编号从 0 到 199, 我们可以考虑使用基于数组的结构来储存病床号与对应病人。例如, 在 Python 中, 我们可以使用一个名字列表, 例如:

```
['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', ...]
* ['雷', '约瑟夫', '珍妮', '乔纳斯', '海伦', '弗吉尼亚', ...]
```

To represent such a list with an array, Python must adhere to the requirement that each cell of the array use the same number of bytes. Yet the elements are strings, and strings naturally have different lengths. Python could attempt to reserve enough space for each cell to hold the *maximum* length string (not just of currently stored strings, but of any string we might ever want to store), but that would be wasteful.

* 要使用数组来表示这样的列表, Python 必须使得数组的每个单元格使用相同数量的字节。然而这些元素是字符串, 字符串自然地具有不同的长度。Python 可以尝试为每个单元格预留足够的空间来容纳最大长度字符串 (不仅仅是当前存储的字符串, 而是我们可能想要存储的任何字符串), 但这将会浪费存储资源。

Instead, Python represents a list or tuple instance using an internal storage mechanism of an array of object *references*. At the lowest level, what is stored is a consecutive sequence of memory addresses at which the elements of the sequence reside. A high-level diagram of such a list is shown in Figure 5.4.

* 相反, Python 使用了一个全新的机制, 它用一个列表储存了所有变量的地址。在最底层, 存储的是连续的序列, 序列元素是内存的地址, 而地址指向的就是实际的元素。这样的列表的高级图示如 5.4 所示。

Although the relative size of the individual elements may vary, the number of bits used to store the memory address of each element is fixed (e.g., 64-bits per address). In this way, Python can support constant-time access to a list or tuple element based on its index.

* 虽然各个实际元素的相对大小可能会变化, 但是用于存储每个元素的存储器地址的位数被固定 (例如, 每个地址 64 位)。这样, Python 可以根据其下标索引支持对列表或元组元素的常量时间复杂的访问。

In Figure 5.4, we characterize a list of strings that are the names of the patients in a hospital. It is more likely that a medical information system would manage more comprehensive information on each patient, perhaps represented as an instance of a Patient class. From the perspective of the list implementation, the same principle applies: The list will simply keep a sequence of references to those objects. Note as well that a reference to the **None** object can be used as an element of the list to represent an empty bed in the hospital.

* 在图 5.4 中, 我们列出了病人姓名的字符串列表。医疗信息系统有可能管理每个患者的更全面的个人信息, 甚至有可能表示为一个 Patient 类。从列表实现的角度来看, 上面叙述的储存地址的数组同样适用这种情形: 列表将简单地存储这些对象的地址。还要注意, 对 None 对象的引用可以表示医院的空床。

The fact that lists and tuples are referential structures is significant to the semantics of these classes. A single list instance may include multiple references to the same object as elements of the list, and it is possible for a single object to be an element of two or more lists, as those lists simply store references back to that object. As an example, when you compute a slice of a list, the result is a new list instance, but that new list has references to the same elements that are in the original list, as portrayed in Figure 5.5.

* 列表和元组可以当作指针结构来用, 这对于它们而言是重要的。一个列表包括对于一个对象的多次引用, 一个对象的地址也可以是多个列表中的元素, 因为这些列表存储的只是对象的地址。例如, 当您计算列表的片段时, 生成的结果是新的列表, 但是新列表依然存储着原始列表的对应元素的地址, 如图 5.5 所示。

When the elements of the list are immutable objects, as with the integer instances in Figure 5.5, the fact that the two lists share elements is not that significant, as neither of the lists can cause a change to the shared object. If, for example, the command `temp[2] = 15` were executed from this configuration, that does not change the existing integer object; it changes the reference in cell 2 of the `temp` list to reference a different object. The resulting configuration is shown in Figure 5.6.

* 当列表的元素是不可变对象时, 与图 5.5 中的整数实例一样, 两个列表共享元素仍然无关紧要, 因为这两个列表都不会导致对象的更改。例如, 如果从该配置执行命令 `temp[2] = 15`, 则不改变现有的整数对象; 它会更改临时列表 `temp[2]` 中的引用以引用其他对象。得到的结果如图 5.6 所示。

The same semantics is demonstrated when making a new list as a copy of an existing one, with a syntax such as `backup = list(primes)`. This produces a new list that is a *shallow copy* (see Section 2.6), in that it references the same elements as in the first list. With immutable elements, this point is moot. If the contents of the list were of a mutable type, a *deep copy*, meaning a new list with *new* elements, can be produced by using the `deepcopy` function from the `copy` module.

* 当使用诸如 `backup = list(primes)` 这样的语法将新的列表作为现有的列表 `copy` 时，会显示相同的语义。这产生了一个浅拷贝（见第 2.6 节）的新列表，因为它引用了与第一个列表中相同的元素。对于不可变的元素，这一点是无法克服的。如果列表的内容是可变类型，则可以通过使用复制模块中的 `deepcopy` 函数来生成深拷贝，即具有新元素的新列表。

As a more striking example, it is a common practice in Python to initialize an array of integers using a syntax such as `counters = [0] * 8`. This syntax produces a list of length eight, with all eight elements being the value zero. Technically, all eight cells of the list reference the *same* object, as portrayed in Figure 5.7.

* 作为一个更引人注目的例子，Python 中常见的做法是使用诸如 `counters = [0] * 8` 的语句来初始化整数数组。该语法生成一个长度为 8 的列表，其中所有八个元素都为零。从技术上看，列表中的所有八个单元格都引用了同一个对象，如图 5.7 所示。

At first glance, the extreme level of aliasing in this configuration may seem alarming. However, we rely on the fact that the referenced integer is immutable. Even a command

such as `counters[2] += 1` does not technically change the value of the existing integer instance. This computes a new integer, with value $0 + 1$, and sets cell 2 to reference the newly computed value. The resulting configuration is shown in Figure 5.8.

* 首先，这个配置中的别名建立可能会令人震惊。但是，我们凭借的是整数是不可变这一事实。即使一个诸如 `counters[2] += 1` 的命令在技术上也不会改变现有整数实例的值。这将计算产生一个新的整数，值为 $0 + 1$ ，并设置单元 2 以引用新计算的值。所得到的配置如图 5.8 所示。

As a final manifestation of the referential nature of lists, we note that the `extend` command is used to add all elements from one list to the end of another list. The extended list does not receive copies of those elements, it receives references to those elements. Figure 5.9 portrays the effect of a call to `extend`.

* 作为列表存储地址这一性质的最终表现，`extend` 命令用于将一个列表的所有元素添加到另一个列表的末尾。扩展列表不会接受到这些元素的副本，它会接收对这些元素的引用。图 5.9 描绘了一个调用的扩展效果。

5.2.2 Compact Arrays in Python

5.3 Dynamic Array and Amortization

When creating a low-level array in a computer system, the precise size of that array must be explicitly declared in order for the system to properly allocate a consecutive piece of memory for its storage. For example, Figure 5.11 displays an array of 12 bytes that might be stored in memory locations 2146 through 2157.

* 在计算机系统中创建低级数组时，必须明确声明该数组的精确大小，以便系统正确分配连续的存储空间。例如，图 5.11 显示了可能存储在存储单元 2146 到 2157 中的 12 个字节的数组。（图略）



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

Because the system might dedicate neighboring memory locations to store other data, the capacity of an array cannot trivially be increased by expanding into subsequent cells. In the context of representing a Python tuple of str instance, this constraint is no problem. Instance of those classes are immutable, so the correct size for an underlying array can be fixed when the object is instantiated.

因为系统可能分配相邻的存储空间给其他的数据，因此数组的容量不能简单地通过往后延伸而扩展容量。在表达 Python 中的元组这个 str 类的实例时，这种约束条件成立。这些类的实例都是不能改变的，因此只有当一个对象被实例化的时候，底层数组的大小才能被确定下来。

Python's list class presents a more interesting abstraction. Although a list has a particular length when constructed, the class allows us to add elements to the list, with no apparent limit on the overall capacity of the list. To provide this abstraction, Python relies on an algorithmic sleight of hand known as a **dynamic array**.

* Python 的列表类很有趣。列表在构造时具有特定的长度，但对列表的整体容量没有明显的限制。为了实现这个，Python 采用了动态数组。

The first key to providing the semantics of a dynamic array is that a list instance maintains an underlying array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to append a new element to the list by using the next available cell of the array.

* 动态数组第一个实现关键是系统保有一个比当前列表长度更大底层数组（也就是实际的内存占用，当前数组是不满的）。例如，当用户可能已经创建了具有五个元素的列表时，系统可能已经预留了能够存储八个对象（而不是仅五个）的底层数组。多余的容量使得使用数组的下一个可用单元格可以轻松地将一个新元素添加到

列表中。

If a user continues to append elements to a list, any reserved capacity will eventually be exhausted. In that case, the class requests a new, larger array from the system, and initializes the new array so that its prefix matches that of the existing smaller array. At that point in time, the old array is no longer needed, so it is reclaimed by the system. Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

* 如果用户继续将元素添加到列表中，则任何预留的容量将最终用尽。在这种情况下，列表会从系统中请求一个新的更大的数组，并初始化新的数组，使其前缀与现有的较小的数组匹配。在这个时间点，旧的阵列已经不再需要了，所以它被系统回收。直观地说，这个策略就像寄居蟹一样，当它的身体长大后，它会移居到更大的壳中。

We give empirical evidence that Python's list class is based upon such a strategy. The source code for our experiment is displayed in Code Fragment 5.1, and a sample output of that program is given in Code Fragment 5.2. We rely on a function named `getsizeof` that is available from the `sys` module. This function reports the number of bytes that are being used to store an object in Python. For a list, it reports the number of bytes devoted to the array and other instance variables of the list, but *not* any space devoted to elements referenced by the list.

* 这里有以上叙述的实证证据。源代码在 5.1 中，而 5.2 中给出了该程序的输出。我们借助 `sys` 模块的 `getsizeof` 的函数。此函数反馈 Python 对象的占用字节数。对于列表，它会反馈数组本身的字节数（即这个储存地址的数组中所有的地址的长度，想想 Python 数组的定义），它不会返回用于列表中实际元素的占用空间。

In evaluating the results of the experiment, we draw attention to the first line of output from Code Fragment 5.2. We see that an empty list instance already requires a certain number of bytes of memory (72 on our system). In fact, each object in Python maintains some state, for example, a reference to denote the class to which it belongs. Although we cannot directly access private instance variables for a list, we can speculate that in some form it maintains state information akin to:

* 在评估实验结果时，注意代码片段 5.2 的第一行输出。空列表已经需要一定数量的内存字节（我们系统上有 72 个）。事实上，Python 中的每个对象都保持一些状态，例如，储存表示它所属的类的变量。虽然我们不能直接访问列表的私有实例变量，但我们可以推测，在某种形式中，它状态信息的保存形式类似于：

_n	The number of actual elements currently stored in the list.
	* 当前存储在列表中的实际元素数。
_capacity	The maximum number of elements that could be stored in the currently allocated array.

* 可以存储在当前分配的数组中的最大元素数。

_A The reference to the currently allocated array (initially None).

* 对当前分配的数组的引用（最初没有）。

As soon as the first element is inserted into the list, we detect a change in the underlying size of the structure. In particular, we see the number of bytes jump from 72 to 104, an increase of exactly 32 bytes. Our experiment was run on a 64-bit machine architecture, meaning that each memory address is a 64-bit number (i.e., 8 bytes). We speculate that the increase of 32 bytes reflects the allocation of an underlying array capable of storing four object references. This hypothesis is consistent with the fact that we do not see any underlying change in the memory usage after inserting the second, third, or fourth element into the list.

* 一旦将第一个元素插入到列表中，我们会检测到结构的底层大小的变化。特别是，我们看到字节数从 72 跳到 104，正好增加了 32 个字节。我们的实验运行在 64 位机器架构上，这意味着每个存储器地址是 64 位数（即 8 字节）。我们推测，32 字节的增加反映了能够存储四个对象。这个假设与将第二个，第三个或第四个元素插入到列表中后，内存使用情况看不到任何潜在的变化是一致的。

After the fifth element has been added to the list, we see the memory usage jump from 104 bytes to 136 bytes. If we assume the original base usage of 72 bytes for the list, the total of 136 suggests an additional $64 = 8 * 8$ bytes that provide capacity for up to eight object references. Again, this is consistent with the experiment, as the memory usage does not increase again until the ninth insertion. At that point, the 200 bytes can be viewed as the original 72 plus an additional 128-byte array to store 16 object references. The 17th insertion pushes the overall memory usage to $272 = 72 + 200 = 72 + 25 * 8$, hence enough to store up to 25 element references.

* 在将第五个元素添加到列表之后，我们看到内存使用率从 104 个字节跳到 136 个字节。如果我们假设列表的

原始基本使用为 72 字节，则总共 136 个表示额外的 $64 = 8 * 8$ 字节，最多可提供八个对象引用的容量。这与实验一致，因为在第九次插入之前，内存使用量不会再增加。在这一点上，200 字节可以被看作是原始的 72 加一个额外的 128 字节，数组存储 16 个对象引用。第 17 次插入将整体内存使用率推送到 $272 = 72 + 200 = 72 + 25 * 8$ ，因此足以存储多达 25 个元素引用。

Because a list is a referential structure, the result of `getsizeof` for a list instance only includes the size for representing its primary structure; it does not account for memory used by the objects that are elements of the list. In our experiment, we repeatedly append `None` to the list, because we do not care about the contents, but we could append any type of object without affecting the number of bytes reported by `getsizeof(data)`.

* 因为列表是一个储存地址的容器，所以列表实例的 `getizeof` 的结果只包括表示其主要结构的大小；它不考虑列表元素对象使用的内存。在我们的实验中，我们反复将 `None` 添加到列表中，因为我们不关心内容，但是我们可以附加任何类型的对象，而不会影响 `getsizeof(data)` 报告的字节数。

If we were to continue such an experiment for further iterations, we might try to discern the pattern for how large of an array Python creates each time the capacity of the previous array is exhausted (see Exercises R-5.2 and C-5.13). Before exploring the precise sequence of capacities used by Python, we continue in this section by describing a general approach for implementing dynamic arrays and for performing an asymptotic analysis of their performance.

* 如果我们继续进行这样一个实验来进一步迭代，我们可能会尝试辨别每次 Python 数组耗尽时数组 Python 的大小（参见练习 R-5.2 和 C-5.13）的模式。在探索 Python 使用的精确的容量顺序之前，我们将继续介绍实现动态数组的一般方法，并对其性能进行渐近分析。

5.3.1 Implementing a Dynamic Array

Although the Python list class provides a highly optimized implementation of dynamic arrays, upon which we rely for the remainder of this book, it is instructive to see how such a class might be implemented.

* 虽然 Python 列表类提供了一个高度优化的动态数组的实现，我们依赖于这本书的其余部分，但是看到究竟是如何实现这样一个类还是有启发性的。

The key is to provide means to grow the array A that stores the elements of a list. Of course, we cannot actually grow that array, as its capacity is fixed. If an element is appended to a list at a time when the underlying array is full, we perform the following steps:

* 提供增加存储列表元素的数组 A 的方法是关键。当然，我们实际上不能增长该数组，因为它的容量是固定的。如果一个元素在底层数组已满时被附加到列表中，我们将执行以下步骤：

1. Allocate a new array B with larger capacity.
* 分配一个容量较大的新数组 B
2. Set $B[i] = A[i]$, for $i = 0, \dots, n-1$, where n denotes current number of items.
* 令 $B[i] = A[i]$, $i = 0, \dots, n-1$, 其中 n 表示当前项目数。
3. Set $A = B$, that is, we henceforth use B as the array supporting the list.
* 令 $A = B$, 以后使用 B 作为数组。
4. Insert the new element in the new array.
* 把新元素插入到新的数组中。

An illustration of this process is shown in Figure 5.12.

* 此过程的图示如 5.12.

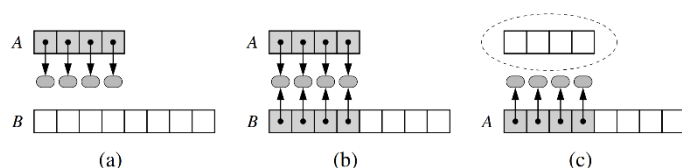


Figure 5.12: An illustration of the three steps for “growing” a dynamic array: (a) create new array B ; (b) store elements of A in B ; (c) reassign reference A to the new array. Not shown is the future garbage collection of the old array, or the insertion of the new element.

The remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled. In Section 5.3.2, we will provide a mathematical analysis to justify such a choice.

* 要考虑的剩余问题是要创建多大的新数组。常使新数组具有现有数组的两倍的容量。在第 5.3.2 节中，我们将提供一个数学的分析来证明这个的选择。

In Code Fragment 5.3, we offer a concrete implementation of dynamic arrays in Python. Our `DynamicArray` class is designed using ideas described in this section. While consistent with the interface of a Python list class, we provide only limited functionality in the form of an `append` method, and accessors `__len__` and `__getitem__`. Support for creating low-level arrays is provided by a module named `ctypes`. Because we will not typically use such a low-level structure in the remainder of this book, we omit a detailed explanation of the `ctypes` module. Instead, we wrap the necessary command for declaring the raw array within a private utility method `_make_array`. The hallmark expansion procedure is performed in our nonpublic `_resize` method.

* 在 Code Fragment 5.3 中，我们提供了 Python 中动态数组的具体实现。我们的 `DynamicArray` 类使用本节中描述的想法进行设计。虽然与 Python 列表类的接口一致，但我们仅提供了 `append` 方法和 `__len__` 与 `__getitem__` 访问等有限的功能。名为 `ctypes` 的模块创建低级数组。因为在本书的其余部分我们通常不会使用这样一个低级别的结构，所以我们省略了 `ctypes` 模块的详细说明。相反，我们包含必要的命令，用于在私有方法 `_make_array` 中声明原始数组。标志扩展程序在我们的非公有 `_resize` 方法中执行。

5.3.2 Amortized Analysis of Dynamic Array

* 动态数组的均摊分析

In this section, we perform a detailed analysis of the running time of operations on dynamic arrays. We use the big-Omega notation introduced in Section 3.3.1 to give an asymptotic lower bound on the running time of an algorithm or step within it.

* 在本节中，我们对动态数组的运行时间进行详细分析。我们使用第 3.3.1 节中介绍的大欧米茄符号来给出其中算法或步骤的运行时间的渐近下界。

The strategy of replacing an array with a new, larger array might at first seem slow, because a single append operation may require $\Omega(n)$ time to perform, where n is the current number of elements in the array. However, notice that by doubling the capacity during an array replacement, our new array allows us to add n new elements before the array must be replaced again. In this way, there are many simple append operations for each expensive one (see Figure 5.13). This fact allows us to show that performing a series of operations on an initially empty dynamic array is efficient in terms of its total running time.

* 用新的更大阵列替换阵列的策略可能很慢，因为单个追加操作可能需要执行 $\Omega(n)$ 个时间，其中 n 是阵列中当前的元素数。然而，请注意，通过在阵列替换期间将容量加倍，我们的新数组允许我们在数组再次替换之前添加 n 个新元素。这样，每次昂贵的扩充操作后可以进行多次简单的操作（见图 5.13）。从这个事实使我们可以看出，在最初的空动态数组上执行一系列操作在其总运行时间方面是有效的。

Using an algorithmic design pattern called **amortization**, we can show that performing a sequence of such append operations on a dynamic array is actually quite efficient. To perform an **amortized analysis**, we use an accounting technique where we view the computer as a coin-operated appliance that requires the payment of one **cyber-dollar** for a constant amount of computing time. When an operation is executed, we should have enough cyber-dollars available in our current “bank account” to pay for that operation’s running time. Thus, the total amount of cyber-dollars spent for any computation will be proportional to the total time spent on that computation. The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

* 通过使用一种称为摊销的算法设计模式，我们可以看到，在动态数组上执行这样的扩充操作实际上是非常有效的。要进行摊销分析，我们需要使用会计学原理，我们将计算机视为投币式电器，需要支付一个赛博货币以保持恒定的计算时间。当一个操作被执行时，我们现在的“银行账户”中应该有足够的赛博货币来支付该操作的运行时间。因此，计算总量将与在该计算上花费的总时间成比例。使用这种分析方法的好处是我们可以刻意过多的操作，节省支出。

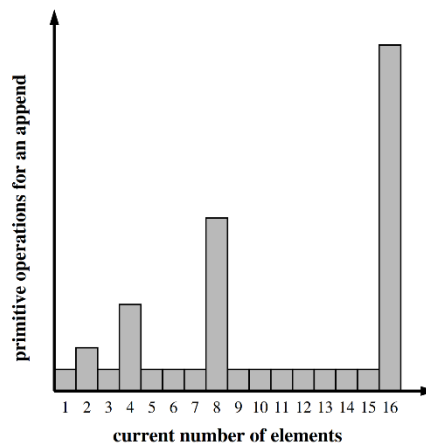


Figure 5.13: Running times of a series of append operations on a dynamic array.

Proposition 5.1: Let S be a sequence implemented by means of a dynamic array with initial capacity one, using the strategy of doubling the array size when full. The total time to perform a series of n append operations in S , starting from S being empty, is $O(n)$.

* 令 S 是具有初始容量 1 的动态数组，使用将数组大小加倍的策略。从 S 开始，在 S 中执行 n 次追加操作的总时间为 $O(n)$ 。

Justification: Let us assume that one cyber-dollar is enough to pay for the execution of each append operation in S , excluding the time spent for growing the array. Also, let us assume that growing the array from size k to size $2k$ requires k cyber-dollars for the time spent initializing the new array. We shall charge each append operation three cyber-dollars. Thus, we overcharge each append operation that does not cause an overflow by two cyber-dollars. Think of the two cyber-dollars profited in an insertion that does not grow the array as being “stored” with the cell in which the element was inserted. An overflow occurs when the array S has 2^i elements, for some integer $i \geq 0$, and the size of the array used by the array representing S is 2^i . Thus, doubling the size of the array will require 2^i cyber-dollars. Fortunately, these cyber-dollars can be found stored in cells 2^{i-1} through $2^i - 1$. (See Figure 5.14.) Note that the previous overflow occurred when the number of elements became larger than 2^{i-1} for the first time, and thus the cyber-dollars stored in cells 2^{i-1} through $2^i - 1$ have not yet been spent. Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of n append operations using $3n$ cyber-dollars. In other words, the amortized running time of each append operation is $O(1)$; hence, the total running time of n append operations is $O(n)$.

* 理由：让我们假设一个赛博钱币足以支付 S 中每个 append 操作的执行，不包括用于增长数组的时间。此外，让我们假设从数组 k 增加到大小 $2k$ 在初始化新数组时需要 k 个赛博钱币。我们将对每个 append 操作收取三个赛博钱币。因此，我们对每个 append 操作进行超量收费，而这不会造成两个赛博钱币的流失。考虑一下，两个在

插入中不会将数组增长为被插入单元格的“存储”。当阵列 S 具有 2^i 个元素，并且由表示 S 的数组所使用的数组的大小为 2^i 时，会发生溢出。因此，将阵列的大小加倍将需要 2^i 的赛博钱币。幸运的是，这些赛博钱币可以存储在 2^{i-1} 到 $2^i - 1$ 的单元格中。（见图 5.14。）请注意，当元件数量第一次变得大于 2^{i-1} 时，发生了先前的过流，因此存储在电池 2^{i-1} 至 $2^i - 1$ 中的网络存储器尚未使用。因此，我们有一个有效的摊销方案，其中每个操作都收取三个赛博钱币，所有的计算时间都被支付。也就是说，我们可以使用 $3n$ 赛博钱币支付执行 n 个追加操作。换句话说，每个追加操作的摊销运行时间为 $O(1)$ ；因此， n 个追加操作的总运行时间为 $O(n)$ 。

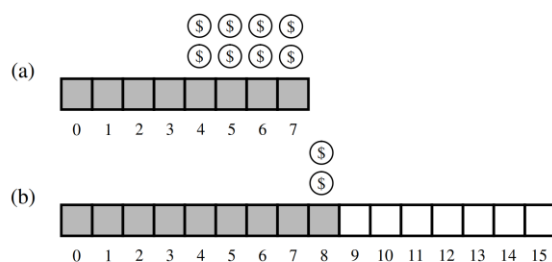


Figure 5.14: Illustration of a series of append operations on a dynamic array: (a) an 8-cell array is full, with two cyber-dollars “stored” at cells 4 through 7; (b) an append operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table. Inserting the new element is paid for by one of the cyber-dollars charged to the current append operation, and the two cyber-dollars profited are stored at cell 8.

【这一段翻译得很混乱，需要仔细看】

Geometric Increase in Capacity

Although the proof of Proposition 5.1 relies on the array being doubled each time we expand, the $O(1)$ amortized bound per operation can be proven for any geometrically increasing progression of array sizes (see Section 2.4.2 for discussion of geometric progressions). When choosing the geometric base, there exists a trade-off between run-time efficiency and memory usage. With a base of 2 (i.e., doubling the array), if the last insertion causes a resize event, the array essentially ends up twice as large as it needs to be. If we instead increase the array by only 25% of its current size (i.e., a geometric base of 1.25), we do not risk wasting as much memory in the end, but there will be more intermediate resize events along the way. Still it is possible to prove an $O(1)$ amortized bound, using a constant factor greater than the 3 cyber-dollars per operation used in the proof of Proposition 5.1 (see Exercise C-5.15). The key to the performance is that the amount of additional space is proportional to the current size of the array.

* 命题 5.1 的证明基于每次扩展时阵列都被加倍，不过对于任何以几何级数增加的数组，可以用摊销法证明每个操作都是 $O(1)$ 时间复杂度（参见第 2.4.2 节讨论几何进度）。在选择几何基数时，需要在运行时效率和内存使用之间权衡。如果使用 2 的基数（即，将数组加倍），那么一旦最后一个插入导致 resize 事件，则该数组基本上会结束它所需的两倍。如果我们将几何基数设置为

1.25，那么我们不会浪费最多的内存，反而是会有更多的大小调整。这仍然有可能被证明是 $O(1)$ 量级的。证明关键在于空间增量与数组当前大小成比例。

Beware of Arithmetic Progression

* 当心等差数列

To avoid reserving too much space at once, it might be tempting to implement a dynamic array with a strategy in which a constant number of additional cells are reserved each time an array is resized. Unfortunately, the overall performance of such a strategy is significantly worse. At an extreme, an increase of only one cell causes each append operation to resize the array, leading to a familiar $1 + 2 + 3 + \dots + n$ summation and $\Omega(n^2)$ overall cost. Using increases of 2 or 3 at a time is slightly better, as portrayed in Figure 5.13, but the overall cost remains quadratic.

* 为了避免一次性保留太多的空间，很有可能需要实现具有策略的动态数组，其中每次调整数组时都保留一定数量的附加单元格。不幸的是，这种策略的整体表现显着更糟。在极端情况下，仅增加一个单元会导致每个追加操作调整数组的大小，导致熟悉的 $1 + 2 + 3 + \dots + n$ 求和 $\Omega(n^2)$ 总成本。如图 5.13 所示，一次使用 2 或 3 的增加稍好一些，但总成本保持二次方。

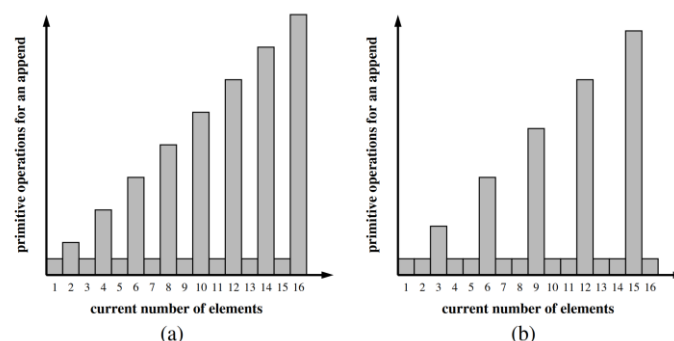


Figure 5.15: Running times of a series of append operations on a dynamic array using arithmetic progression of sizes. (a) Assumes increase of 2 in size of the array, while (b) assumes increase of 3.

Using a *fixed* increment for each resize, and thus an arithmetic progression of intermediate array sizes, results in an overall time that is quadratic in the number of operations, as shown in the following proposition. Intuitively, even an increase in 1000 cells per resize will become insignificant for large data sets.

* 由于每次 resize 使用固定增量，因此使用中间数组大小的算术进程，导致操作次数中的二次方的总时间，如下面的命题所示。直观地说，即使每次 resize 的增量是 1000 个单元，这对于大型数据集也是毫无意义的。

Proposition 5.2: Performing a series of n append operations on an initially empty dynamic array using a fixed increment with each resize takes $\Omega(n^2)$ time.

* 命题 5.2: 在最初的空动态数组上执行 n 次操作，如果每次 resize 都使用固定增量，那么时间复杂度是 $\Omega(n^2)$ 。

Justification: Let $c > 0$ represent the fixed increment in capacity that is used for each resize event. During the series of n append operations, time will have been spent initializing arrays of size $c, 2c, 3c, \dots, mc$ for $m = \lfloor n/c \rfloor$, and therefore, the overall time would be proportional to $c + 2c + 3c + \dots + mc$. By Proposition 3.3, this sum is

* 证明: 令 $c > 0$ 表示用于每个 `resize` 事件的固定增量。在 n 个 `append` 操作期间, 将花费初始化时间分别为 $c, 2c, 3c, \dots, mc$ ($m = \lfloor n/c \rfloor$), 因此, 总时间将与 $c + 2c + 3c + \dots + mc$ 成正比。通过命题 3.3, 这个总和是

$$\sum_{i=1}^m ci = c \cdot \sum_{i=1}^m i = c \frac{m(m+1)}{2} \geq c \frac{\frac{n}{c} \left(\frac{n}{c} + 1 \right)}{2} \geq \frac{n^2}{2c}$$

Therefore, performing the n append operations takes $\Omega(n^2)$ time.

* 因此, 进行 n 次 `append` 操作的时间复杂度为 $\Omega(n^2)$ 。

A lesson to be learned from Propositions 5.1 and 5.2 is that a subtle difference in an algorithm design can produce drastic differences in the asymptotic performance, and that a careful analysis can provide important insights into the design of a data structure.

* 从命题 5.1 和 5.2 中学到的教训是, 算法设计中的微妙差异可能在渐近性能中产生巨大差异, 仔细分析可以为数据结构的设计提供重要指导。

Memory Usage and Shrinking an Array

* 内存占用与收缩数组

Another consequence of the rule of a geometric increase in capacity when appending to a dynamic array is that the final array size is guaranteed to be proportional to the overall number of elements. That is, the data structure uses $O(n)$ memory. This is a very desirable property for a data structure.

* 动态数组按几何级数增加, 这回使得最终的数组大小保证与元素的总数成正比。也就是说, 数据结构占用的

内存是 $O(n)$ 量级的。这是数据结构非常理想的属性。

If a container, such as a Python list, provides operations that cause the removal of one or more elements, greater care must be taken to ensure that a dynamic array guarantees $O(n)$ memory usage. The risk is that repeated insertions may cause the underlying array to grow arbitrarily large, and that there will no longer be a proportional relationship between the actual number of elements and the array capacity after many elements are removed.

* 如果容器 (如 Python 列表) 提供了删除一个或多个元素的操作, 则必须更加注意, 确保动态数组是 $O(n)$ 内存占用。风险是重复的插入可能导致底层数组任意增长, 并且在删除多个元素之后, 实际元素数与阵列容量之间将不再具有比例关系。

A robust implementation of such a data structure will shrink the underlying array, on occasion, while maintaining the $O(1)$ amortized bound on individual operations. However, care must be taken to ensure that the structure cannot rapidly oscillate between growing and shrinking the underlying array, in which case the amortized bound would not be achieved. In Exercise C-5.16, we explore a strategy in which the array capacity is halved whenever the number of actual element falls below one fourth of that capacity, thereby guaranteeing that the array capacity is at most four times the number of elements; we explore the amortized analysis of such a strategy in Exercises C-5.17 and C-5.18.

* 有时这种数据结构的强大实现会缩小基础阵列, 同时保持对个别操作的 $O(1)$ 摊销限制。然而, 必须注意确保结构在扩充和收缩底层阵列之间不能迅速振荡, 在这种情况下不能实现摊销。在练习 C-5.16 中, 我们探讨一种策略, 其中当实际元素的数量低于该容量的四分之一时, 阵列容量减半, 从而保证阵列容量是元素数量的四倍以上。我们在练习 C-5.17 和 C-5.18 中探讨了这种策略的摊销分析。

5.3.3 Python’s List Class

* Python 的列表类

The experiments of Code Fragment 5.1 and 5.2, at the beginning of Section 5.3, provide empirical evidence that Python’s list class is using a form of dynamic arrays for its storage. Yet, a careful examination of the intermediate array capacities (see Exercises R-5.2 and C-5.13) suggests that Python is not using a pure geometric progression, nor is it using an arithmetic progression.

* 在第 5.3 节开头的 Code Fragment 5.1 和 5.2 的实验提供了实验证明，Python 的列表类使用一种形式的动态数组。然而，仔细检查中间阵列容量（参见练习 R-5.2 和 C-5.13）表明 Python 不使用纯粹的几何级数，也不使用算术进程。

With that said, it is clear that Python’s implementation of the append method exhibits amortized constant-time behavior. We can demonstrate this fact experimentally. A single append operation typically executes so quickly that it would be difficult for us to accurately measure the time elapsed at that granularity, although we should notice some of the more expensive operations in which a resize is performed. We can get a more accurate measure of the amortized cost per operation by performing a series of n append operations on an initially empty list and determining the *average* cost of each. A function to perform that experiment is given in Code Fragment 5.4.

* 很明显，Python 对 append 方法的实现表现出摊销的恒定时间行为。我们可以通过实验证明这一点。单个追加操作通常执行得很快，因此我们很难准确地测量在该进

程下所经过的时间，尽管我们应该注意到执行调整大小的一些更昂贵的操作。我们可以通过在最初空的列表上执行一系列的 append 操作，并确定每个操作的平均成本，从而获得对每个操作的摊销成本的更精确的测量。在代码片段 5.4 中给出了执行该实验的功能。

Technically, the time elapsed between the start and end includes the time to manage the iteration of the for loop, in addition to the append calls. The empirical results of the experiment, for increasingly large values of n , are shown in Table 5.2. We see higher average cost for the smaller data sets, perhaps in part due to the overhead of the loop range. There is also natural variance in measuring the amortized cost in this way, because of the impact of the final resize event relative to n . Taken as a whole, there seems clear evidence that the amortized time for each append is independent of n .

* 从技术上讲，开始和结束之间经过的时间包括管理 for 循环的迭代以及 append 调用的时间。对于越来越大的 n 值，实验的经验结果如表 5.2 所示。我们看到较小的数据集的平均成本更高，这可能部分是由于循环范围的开销。由于最终调整大小事件相对于 n 的影响，在这种方式下测量均摊成本也存在自然差异。总的来说，似乎有明确的证据表明，每个附加物的摊销时间与 n 无关。

n	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
μs	0.219	0.158	0.164	0.151	0.147	0.147	0.149

Table 5.2: Average running time of append, measured in microseconds, as observed over a sequence of n calls, starting with an empty list.

5.4 Efficiency of Python’s Sequence Types

In the previous section, we began to explore the underpinnings of Python’s list class, in terms of implementation strategies and efficiency. We continue in this section by examining the performance of all of Python’s sequence types.

* 在上一节中，我们开始探索 Python 列表类的基础，主要研究方向是实现策略和效率方面。我们通过研究 Python 所有的序列类型的性能来继续本节的内容。

5.4.1 Python’s List and Tuple Classes

* Python 的列表类与元组类

The *nonmutating* behaviors of the list class are precisely those that are supported by the tuple class. We note that tuples are typically more memory efficient than lists because they are immutable; therefore, there is no need for an underlying dynamic array with surplus capacity. We summarize the asymptotic efficiency of the nonmutating behaviors of the list and tuple classes in Table 5.3. An explanation of this analysis follows.

* 列表类的不显示行为正是那些由元组支持的行为。我们注意到，元组因为不可更改而通常会比列表更有效率；因此，元组不需要具有额外容量的底层动态数组。我们总结了列表和元组的行为区别。以下是对此分析的解释。

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><</code> , <code><=</code> , <code>></code> , <code>>=</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 $k = \min(n_1, n_2)$.

Constant-Time Operations

* 常量级操作

The length of an instance is returned in constant time because an instance explicitly maintains such state information. The constant-time efficiency of syntax `data[j]` is assured by the underlying access into an array.

* 计算列表的长度是常量级操作，因为列表实例直接把这个数值作为状态信息进行了保存。语句 `data[j]` 的常量级效率是对底层数组进行访问而实现的。

Searching for Occurrences of a Value

* 查找数值元素

Each of the `count`, `index`, and `__contains__` methods proceed through iteration of the sequence from left to right. In fact, Code Fragment 2.14 of Section 2.4.3 demonstrates how those behaviors might be implemented. Notably, the loop for computing the count must proceed through the entire sequence, while the loops for checking containment of an element or determining the index of an element immediately exit once they find the leftmost occurrence of the desired value, if one exists. So while `count` always examines the n elements of the sequence, `index` and `__contains__` examine n elements in the worst case, but may be faster. Empirical evidence can be found by setting `data = list(range(10000000))` and then comparing the relative efficiency of the test, `5 in data`, relative to the test, `9999995 in data`, or even the failed test, `-5 in data`.

* `count`，`index` 和 `__contains__` 方法中的每一个通过从左到右的顺序进行迭代。实际上，2.4.3 节的 Code Fragment 2.14 演示了如何实现这些行为。值得注意的是，用于计数的循环必须通过整个序列进行，而用于检查的循环或确定元素索引的循环，将在得到结果之后立即退出，。所以 `count` 总是检查序列的 n 个元素，而 `index` 和 `__contains__` 只有在最坏的情况下才检查 n 个元素，所以后两者可能会更快。通过令 `data = list(range(10000000))`，然后比较数据的相对效率，检索 `5`，`9999995`，甚至是不存在的 `-5`，都会有不一样的运行时间。

Lexicographic Comparisons

* 字典比较

Comparisons between two sequences are defined lexicographically. In the worst case, evaluating such a condition requires an iteration taking time proportional to the length of the shorter of the two sequences (because when one sequence ends, the lexicographic result can be determined). However, in some cases the result of the test can be evaluated more efficiently. For example, if evaluating `[7, 3, ...] < [7, 5, ...]`, it is clear that the result is `True` without examining the remainders of those lists, because the second element of the left operand is strictly less than the second element of the right operand.

* 两个序列之间的比较是按字典顺序定义的。在最坏的情况下，这样的比较需要比完长度较小的那个序列。然而，在某些情况下，可以更有效地评估测试结果。例如，如果评估 `[7,3,...] < [7,5,...]`，很清楚，（到第二个元素的时候）结果就是 `True`，而这之后不检查这些列表的余数，因为左序列的第二个元素是严格小于右序列的第二个元素。

Creating New Instances

* 创建新实例

The final three behaviors in Table 5.3 are those that construct a new instance based on one or more existing instances. In all

cases, the running time depends on the construction and initialization of the new result, and therefore the asymptotic behavior is proportional to the length of the result. Therefore, we find that slice `data[6000000:6000008]` can be constructed almost immediately because it has only eight elements, while slice `data[6000000:7000000]` has one million elements, and thus is more time-consuming to create.

* 表 5.3 中的最后三个行为是基于一个或多个现有实例构建新实例的行为。在任何情况下, 运行时间取决于新结果的构建和初始化, 所以消耗时间与返回对象的长度成比例。因此我们发现片段数据[6000000: 6000008]可以几乎立即构造, 因为它只有八个元素, 而片数据[6000000: 7000000]具有一百万个元素, 因此创建更耗时。

Mutating Behaviors

* 更改操作

The efficiency of the mutating behaviors of the list class are described in Table 5.3. The simplest of those behaviors has syntax `data[j] = val`, and is supported by the special `__setitem__` method. This operation has worst-case $O(1)$ running time because it simply replaces one element of a list with a new value. No other elements are affected and the size of the underlying array does not change. The more interesting behaviors to analyze are those that add or remove elements from the list.

* 表 5.3 中描述了列表类的更改行为的有效性。最简单的行为就是语句 `data[j] = val`, 并且由特殊 `__setitem__` 方法支持。这个操作有最坏的 $O(1)$ 运行时间, 因为它只是用一个新的值替换一个列表的一个元素。没有其他元素受到影响, 底层数组的大小也不会改变。要分析的更有趣的行为是从列表中添加或删除元素的行为。

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.

Adding Elements to a List

* 添加元素

In Section 5.3 we fully explored the `append` method. In the

worst case, it requires $\Omega(n)$ time because the underlying array is resized, but it uses $O(1)$ time in the amortized sense. Lists also support a method, with signature `insert(k, value)`, that inserts a given value into the list at index $0 \leq k \leq n$ while shifting all subsequent elements back one slot to make room. For the purpose of illustration, Code Fragment 5.5 provides an implementation of that method, in the context of our `DynamicArray` class introduced in Code Fragment 5.3. There are two complicating factors in analyzing the efficiency of such an operation. First, we note that the addition of one element may require a resizing of the dynamic array. That portion of the work requires $\Omega(n)$ worst-case time but only $O(1)$ amortized time, as per `append`. The other expense for `insert` is the shifting of elements to make room for the new item. The time for that process depends upon the index of the new element, and thus the number of other elements that must be shifted. That loop copies the reference that had been at index $n - 1$ to index n , then the reference that had been at index $n - 2$ to $n - 1$, continuing until copying the reference that had been at index k to $k + 1$, as illustrated in Figure 5.16. Overall this leads to an amortized $O(n - k + 1)$ performance for inserting at index k .

* 在 5.3 节中, 我们充分探讨了 `append` 方法。在最坏的情况下, 它需要 $\Omega(n)$ 时间, 因为底层数组被调整大小, 但是以摊销方式使用 $O(1)$ 时间。列表还支持一种具有签名 `insert(k,value)` 的方法, 将索引 $0 \leq k \leq n$ 的给定值插入到列表中, 同时将所有后续元素移回一个单元腾出空间。为了说明目的, Code Fragment 5.5 在 Code Fragment 5.3 中介绍的 `DynamicArray` 类的上下文中提供了该方法的实现。分析这种操作的效率有两个复杂因素。首先, 我们注意到添加一个元素可能需要调整动态数组的大小。这部分工作需要 $\Omega(n)$ 最坏情况时间, 但只有 $O(1)$ 的摊销时间。`insert` 的其他开销是移动元素为新项目腾出空间。该过程的时间取决于新元素的下标, 即由此取决于必须移动的其他元素的数量。该循环将下标在 $n-1$ 处的指向元素复制到下表 n , 然后将下表为 $n-2$ 的元素移动到 $n-1$ 处, 直到将下标 k 处的指向元素复制到 $k + 1$, 如图 5.16 所示。总体来说, 这导致在索引 k 处插入的摊销 $O(n - k + 1)$ 性能。

When exploring the efficiency of Python's `append` method in Section 5.3.3, we performed an experiment that measured the average cost of repeated calls on varying sizes of lists (see Code Fragment 5.4 and Table 5.2). We have repeated that experiment with the `insert` method, trying three different access patterns:

* 在第 5.3.3 节探讨 Python `append` 方法的效率时, 我们进行了一个实验, 测量了不同大小的列表中重复调用的平均成本 (见代码片段 5.4 和表 5.2)。我们用 `insert` 方法重复了这个实验, 尝试了三种不同的模式:

- In the first case, we repeatedly insert at the beginning of a list,

* 在第一种情况下, 我们反复在列表的开头插入,

```
for n in range(N):
    data.insert(0, None)
```

- In a second case, we repeatedly insert near the middle of a list,

* 在第二种情况下，我们反复在列表中间插入

```
for n in range(N):
    data.insert(n // 2, None)
```

- In a third case, we repeatedly insert at the end of the list,

* 在第三种情况下，我们反复在列表尾端插入

```
for n in range(N):
    data.insert(n, None)
```

The results of our experiment are given in Table 5.5, reporting the *average* time per operation (not the total time for the entire loop). As expected, we see that inserting at the beginning of a list is most expensive, requiring linear time per operation. Inserting at the middle requires about half the time as inserting at the beginning, yet is still $\Omega(n)$ time. Inserting at the end displays $O(1)$ behavior, akin to `append`.

* 我们的实验结果在表 5.5 中给出，报告每个操作的平均时间（而不是整个循环的总时间）。如预期的那样，我们看到，在列表开头插入是最昂贵的，每次操作需要线性时间。在中间插入需要大约一半的时间插入开始，但仍然是 $\Omega(n)$ 时间。最后插入显示 $O(1)$ 行为，类似于 `append`。

	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).

Removing Elements from a List

* 移除元素

Python's list class offers several ways to remove an element from a list. A call to `pop()` removes the last element from a list. This is most efficient, because all other elements remain in their original location. This is effectively an $O(1)$ operation, but the bound is amortized because Python will occasionally shrink the underlying dynamic array to conserve memory.

* Python 的列表类提供了从列表中删除元素的几种方法。对 `pop()` 的调用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原始位置。这实际上是一个 $O(1)$ 操作，但是边界是摊销的，因为 Python 会偶尔收缩基础动态数组以节省内存。

The parameterized version, `pop(k)`, removes the element

that is at index $k < n$ of a list, shifting all subsequent elements leftward to fill the gap that results from the removal. The efficiency of this operation is $O(n - k)$, as the amount of shifting depends upon the choice of index k , as illustrated in Figure 5.17. Note well that this implies that `pop(0)` is the most expensive call, using $\Omega(n)$ time. (see experiments in Exercise R-5.8.)

* 参数化版本 `pop(k)` 删除列表中下标 $k < n$ 的元素，将所有后续元素向左移动，以清除由删除导致的差距。此操作的复杂度为 $O(n-k)$ ，因为移位量取决于下标 k 的选择，如图 5.17 所示。请注意，这意味着 `pop(0)` 是复杂度为 $\Omega(n)$ 最昂贵的调用。（参见练习 R-5.8 中的实验。）

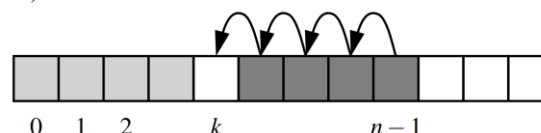


Figure 5.17: Removing an element at index k of a dynamic array.

The list class offers another method, named `remove`, that allows the caller to specify the *value* that should be removed (not the *index* at which it resides). Formally, it removes only the first occurrence of such a value from a list, or raises a `ValueError` if no such value is found. An implementation of such behavior is given in Code Fragment 5.6, again using our `DynamicArray` class for illustration.

* 列表类提供了另一种名为 `remove` 的方法，该方法允许调用者指定应该删除的值（而不是它所驻留的索引）。正式地，它从列表中仅删除第一次出现的值，如果没有找到这样的值，则会引发 `ValueError`。代码片段 5.6 中给出了这种行为的实现，再次使用我们的 `DynamicArray` 类进行说明。

Interestingly, there is no “efficient” case for `remove`; every call requires $\Omega(n)$ time. One part of the process searches from the beginning until finding the value at index k , while the rest iterates from k to the end in order to shift elements leftward. This linear behavior can be observed experimentally (see Exercise C-5.24).

* 有趣的是，没有“有效”的移除案例；每个调用都需要 $\Omega(n)$ 的时间。过程的一部分从一开始就搜索到确定索引 k 处的值，而其余部分从 k 循环到结束，以向左移位元素。这种线性行为可以通过实验观察（参见练习 C-5.24）。

Extending a List

* 扩充数组

Python provides a method named `extend` that is used to add all elements of one list to the end of a second list. In effect, a call to `data.extend(other)` produces the same outcome as the code,

* Python 提供了一个名为 `extend` 的方法，用于将一个列表的所有元素添加到第二个列表的末尾。实际上，与 `data.extend(other)` 产生相同的结果的代码如下所示，

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

In either case, the running time is proportional to the length of the other list, and amortized because the underlying array for the first list may be resized to accommodate the additional elements.

* 在任一情况下，运行时间与 other 列表的长度成比例，并且由于第一列表的底层数组可能被调整大小以适应附加元素而进行分摊。

In practice, the extend method is preferable to repeated calls to append because the constant factors hidden in the asymptotic analysis are significantly smaller. The greater efficiency of extend is threefold. First, there is always some advantage to using an appropriate Python method, because those methods are often implemented natively in a compiled language (rather than as interpreted Python code). Second, there is less overhead to a single function call that accomplishes all the work, versus many individual function calls. Finally, increased efficiency of extend comes from the fact that the resulting size of the updated list can be calculated in advance. If the second data set is quite large, there is some risk that the underlying dynamic array might be resized multiple times when using repeated calls to append. With a single call to extend, at most one resize operation will be performed. Exercise C-5.22 explores the relative efficiency of these two approaches experimentally.

* 实际上，由于常数因子显著较小，所以扩展方法优于重复调用。扩展效率更高是三重的。首先，使用适当的 Python 方法总是有一些优势，因为这些方法通常以编译语言（而不是解释的 Python 代码）实现。第二，完成所有工作的单个函数调用与许多单独的函数调用相比，开销较少。最后，扩展效率的提高来自于可以提前计算更新列表的最终大小。如果第二个数据集相当大，那么在使用重复调用附加时，底层动态数组可能会被调整多次。通过单次调用来扩展，最多只能执行一次调整大小的操作。练习 C-5.22 通过实验探讨了这两种方法的相对有效性。

Constructing New Lists

* 建立新列表

There are several syntaxes for constructing new lists. In almost all cases, the asymptotic efficiency of the behavior is linear in the length of the list that is created. However, as with the case in the preceding discussion of extend, there are significant differences in the practical efficiency.

* 有几种构造新列表的命令。在几乎所有情况下，运行时间的增长幅度在创建的列表的长度上是线性的。然而，与上述 extend 讨论中的情况一样，实际效率存在显著差异。

Section 1.9.2 introduces the topic of *list comprehension*, using an example such as `squares = [k * k for k in range(1, n+1)]` as a shorthand for

* 第 1.9.2 节介绍了列表解析的主题，使用例如 `square = [k * k for k in range(1, n + 1)]` 以下代码的简写

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

Experiments should show that the list comprehension syntax is significantly faster than building the list by repeatedly appending (see Exercise C-5.23).

* 实验会证明，第一种语法比通过反复 append 快得多（参见练习 C-5.23）。

Similarly, it is a common Python idiom to initialize a list of constant values using the multiplication operator, as in `[0] * n` to produce a list of length n with all values equal to zero. Not only is this succinct for the programmer; it is more efficient than building such a list incrementally.

* 类似地，Python 使用乘法运算符来初始化常量值列表，如 `[0] * n` 中所示，产生一个长度为 n 的列表，其中所有值都等于零。这不仅仅是程序员的简洁爱好所导致；实际上这样做更有效。

5.4.2 Python's String Class

* 5.4.2 节 Python 的字符串类

Strings are very important in Python. We introduced their use in Chapter 1, with a discussion of various operator syntaxes in Section 1.3. A comprehensive summary of the named methods of the class is given in Tables A.1 through A.4 of Appendix A. We will not formally analyze the efficiency of each of those behaviors in this section, but we do wish to comment on some notable issues. In general, we let n denote the length of a string. For operations that rely on a second string as a pattern, we let m denote the length of that pattern string.

* 字符串在 Python 中非常重要。我们在第 1 章介绍了它们的用法，并讨论了第 1.3 节中的各种操作符语法。在附录 A 的表 A.1 至 A.4 中给出了类的命名方法的全面总结。我们不会正式分析本节中每个行为的效率，但我们希望对一些值得注意的评论的问题。一般来说，我们让 n 表示字符串的长度。对于依赖于第二个字符串作为参数的操作，我们让 m 表示该 pattern 字符串的长度。

The analysis for many behaviors is quite intuitive. For example, methods that produce a new string (e.g., `capitalize`, `center`, `strip`) require time that is linear in the length of the string that is produced. Many of the behaviors that test Boolean conditions of a string (e.g., `islower`) take $O(n)$ time, examining all n characters in the worst case, but short circuiting as soon as the answer becomes evident (e.g., `islower` can immediately return `False` if the first character is uppercased). The comparison operators (e.g., `==`, `<`) fall into this category as well.

* 对许多行为的分析是非常直观的。例如，产生新字符串（例如 `capitalize`, `center`, `strip`）的方法的复杂度是关于要产生的字符串长度的线性函数。测试字符串的布尔条件（例如，`islower`）的许多行为的复杂度是 $O(n)$ ，在最坏的情况下检查所有 n 个字符，但一旦答案变得明显，立即中断操作（例如，如果第一个字符是大写，`islower` 可以立即返回 `False`，）。比较运算符（例如，`==`, `<`）也属于该类别。

Pattern Matching

* 模式匹配

Some of the most interesting behaviors, from an algorithmic point of view, are those that in some way depend upon finding a string pattern within a larger string; this goal is at the heart of methods such as `__contains__`, `find`, `index`, `count`, `replace`, and `split`. String algorithms will be the topic of Chapter 13, and this particular problem known as **pattern matching** will be the focus of Section 13.2. A naive implementation runs in $O(mn)$ time case, because we consider the $n - m + 1$ possible starting indices for the pattern, and we spend $O(m)$ time at each starting position, checking if the pattern matches. However, in Section 13.2, we will develop an algorithm for finding a pattern of length m within a longer string of length n in $O(n)$ time.

* 从算法的角度来看，一些最有趣的行为是判断大字符串中是否含有小字符串；这个设计目标是以下方法的核心，如 `__contains__`，`find` 查找，`index` 索引，`count` 计数，`replace` 替换和 `split` 拆分。字符串算法将是第 13 章的主题，模式匹配问题将成为第 13.2 节的重点。由于我们考虑了模式的 $n - m + 1$ 个可能的起始索引，我们在每个起始位置花费 $O(m)$ 个时间，检查模式是否匹配，因此在 $O(mn)$ 时间段中运行一个初始实现。然而，在第 13.2 节中，我们将设计一种在复杂度为 $O(n)$ 的算法，其中 n 为 `text` 字符串的长度。

Composing Strings

* 字符串连接

Finally, we wish to comment on several approaches for composing large strings. As an academic exercise, assume that we have a large string named `document`, and our goal is to produce a new string, `letters`, that contains only the alphabetic characters of the original string (e.g., with spaces, numbers, and punctuation removed). It may be tempting to compose a result through repeated concatenation, as follows.

* 最后，我们要对链接字符串的集中方法进行注解。作为一个学术性练习，假设我们有一个大的字符串名为 `document`，我们的目标是生成一个名为 `letters` 新的字符串，这个字符串只包含 `document` 字符串的字母字符（例如，把空格，数字和标点符号都删掉）。通过重复连接来组合结果可能是不错的，如下：

```
# WARNING: do not do this
letters = ""      # start with empty string
for c in document:
    if c.isalpha():
        letters += c      # concatenate alphabetic character
```

While the preceding code fragment accomplishes the goal, it may be terribly inefficient. Because strings are immutable, the command, `letters += c`, would presumably compute the concatenation, `letters + c`, as a new string instance and then reassign the identifier, `letters`, to that result. Constructing that new string would require time proportional to its length. If the final result has n characters, the series of concatenations would take time proportional to the familiar sum $1 + 2 + 3 + \dots + n$, and therefore $O(n^2)$ time.

* 虽然前面的代码片段实现了目标，但可能是非常不合格的。因为字符串是不可变的，所以命令 `letters += c` 的运行，会产生一个新的字符串实例 `letters + c`，然后将该标识符，字母重新分配给该结果。构造新的字符串将需要与其长度成比例的时间。如果最终结果具有 n 个字符，则连续的合并将与合式 $1 + 2 + 3 + \dots + n$ 成比例，即导致 $O(n^2)$ 的时间复杂度。

Inefficient code of this type is widespread in Python, perhaps because of the somewhat natural appearance of the code,

and mistaken presumptions about how the `+=` operator is evaluated with strings. Some later implementations of the Python interpreter have developed an optimization to allow such code to complete in linear time, but this is not guaranteed for all Python implementations. The optimization is as follows. The reason that a command, `letters += c`, causes a new string instance to be created is that the original string must be left unchanged if another variable in a program refers to that string. On the other hand, if Python knew that there were no other references to the string in question, it could implement `+=` more efficiently by directly mutating the string (as a dynamic array). As it happens, the Python interpreter already maintains what are known as reference counts for each object; this count is used in part to determine if an object can be garbage collected. (See Section 15.1.2.) But in this context, it provides a means to detect when no other references exist to a string, thereby allowing the optimization.

* 这种类型的无效代码在 Python 中很普遍，也许是由于代码看起来很自然，而且很有可能程序员并不懂 `+=` 运算符的实际内部操作。一些后来的 Python 解释器的实现已经开发了一个优化，以允许这样的代码在线性时间内完成，但是这并不能保证所有的 Python 程序。优化如下。语句 `+= c` 会创建新的字符串实例，就是为了保证其他引用 `c` 的变量不会发生紊乱。另一方面，如果 Python 知道没有其他对该字符串的引用，则可以通过直接变换字符串（作为动态数组）来实现扩充。就这样，Python 解释器已经保留了每个对象的引用计数；这个计数部分用于确定对象是否可以被垃圾回收。（见第 15.1.2 节）在这种情况下，Python 解释器提供了一种方法来检测字符串有没有被引用，这样一来就可以优化了。

A more standard Python idiom to guarantee linear time composition of a string is to use a temporary list to store individual pieces, and then to rely on the `join` method of the `str` class to compose the final result. Using this technique with our previous example would appear as follows:

* 确保字符串在线性时间内组合的更标准做法是使用临时列表来存储单个片段，然后依靠 `str` 类的 `join` 方法来构成最终结果。实例如下：

```
temp = [] # start with empty list
for c in document:
    if c.isalpha():
        temp.append(c) # append alphabetic character
letters = ''.join(temp) # compose overall result
```

This approach is guaranteed to run in $O(n)$ time. First, we note that the series of up to n `append` calls will require a total of $O(n)$ time, as per the definition of the amortized cost of that operation. The final call to `join` also guarantees that it takes time that is linear in the final length of the composed string.

* 这种方法保证在 $O(n)$ 时间内运行。首先，我们注意到，根据该操作的摊销成本的定义，一系列 n 个 `append` 调用将需要总共 $O(n)$ 时间。最后的 `join` 函数调用也保证在最后组合字符串的时候需要线性时间复杂度。

As we discussed at the end of the previous section, we can further improve the practical execution time by using a list comprehension syntax to build up the temporary list, rather than by repeated calls to `append`. That solution appears as,

* 正如我们在上一节结尾讨论的，我们可以通过使用列表解析语法来构建临时列表来进一步改进实际的执行时间，而不是通过重复的追加调用。代码如下

```
letters = ''.join([c for c in document if c.isalpha()])
```

Better yet, we can entirely avoid the temporary list with a generator comprehension:

* 更好的是，我们可以使用生成器来完全避免临时列表：

```
letters = ''.join(c for c in document if c.isalpha())
```

5.5 Using Array-Based Sequence

* 对基于数组的序列的调用

5.5.1 Sorting High Scores for a Game

* 对游戏高分榜进行排序

The first application we study is storing a sequence of high score entries for a video game. This is representative of many applications in which a sequence of objects must be stored. We could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data-structuring concepts.

* 我们研究的第一个应用是为电子游戏存储一系列高分记录。这是许多应用程序的代表，其中必须存储一系列对象。我们可以很容易地选择为医院病人或足球队的球员名字存储记录。然而，我们专注于存储高分条目，这是一个已经足够丰富的简单应用程序，可以呈现一些重要的数据结构概念。

To begin, we consider what information to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we identify as `_score`. Another useful thing to include is the name of the person earning this score, which we identify as `_name`. We could go on from here, adding fields representing the date the score was earned or game statistics that led to that score. However, we omit such details to keep our example simple. A Python class, `GameEntry`, representing a game entry, is given in Code Fragment 5.7.

* 首先，我们考虑在表示高分记录时要包含哪些信息。显然，要用一个整数表示分数，我们将之命名为 `_score`。另外一个就是获得这个分数的人的名字，我们将其标识为 `_name`。我们可以从这里继续，添加代表得分的日期的字段或导致该分数的游戏统计。但是，我们省略了这样的细节，以保持我们的例子简单。代码 5.7 中给出了一个表示游戏条目的 Python 类 `GameEntry`。

A Class for High Scores

* 记录高分的 Python 类

To maintain a sequence of high scores, we develop a class named `Scoreboard`. A scoreboard is limited to a certain number of high scores that can be saved; once that limit is reached, a new score only qualifies for the scoreboard if it is strictly higher than the lowest “high score” on the board. The length of the desired scoreboard may depend on the game, perhaps 10, 50, or 500. Since that limit may vary depending on the game, we allow it to be specified as a parameter to our `Scoreboard` constructor.

* 为了保存高分的序列，我们开发了一个名为“记分榜”的类。记分牌限于固定数量的高分；如果一个新的得分只有严格大于榜上最低分的情况下才能上榜。所需记分板的长度可能取决于游戏，可能是 10、50 或 500。由于

该限制可能因游戏而异，因此我们将其指定为我们的 `Scoreboard` 类的构造函数的参数。

Internally, we will use a Python list named `board` in order to manage the `GameEntry` instances that represent the high scores. Since we expect the score-board to eventually reach full capacity, we initialize the list to be large enough to hold the maximum number of scores, but we initially set all entries to `None`. By allocating the list with maximum capacity initially, it never needs to be resized. As entries are added, we will maintain them from highest to lowest score, starting at index 0 of the list. We illustrate a typical state of the data structure in Figure 5.18.

* 在内部，我们将使用名为 `_board` 的 Python 列表来管理代表高分的 `GameEntry` 实例。由于我们预计积分榜最终将达到全部容量，因此我们将该列表初始长度设置到足够大以保持最大分数，我们最初将所有条目设置 `None`。通过最初分配容量的列表，它不需要调整大小。随着条目被添加，我们将从列表的第 0 元素开始，将它们从最高到最低分保持一致。如图 5.18。

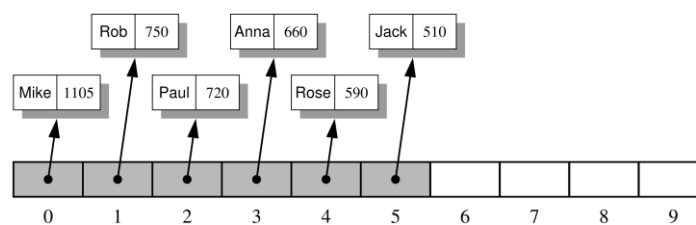


Figure 5.18: An illustration of an ordered list of length ten, storing references to six `GameEntry` objects in the cells from index 0 to 5, with the rest being `None`.

A complete Python implementation of the `Scoreboard` class is given in Code Fragment 5.8. The constructor is rather simple. The command

* 在 Code Fragment 5.8 中给出了 `Scoreboard` 类的完整 Python 实现。构造函数相当简单。命令

```
self.board = [None] * capacity
```

creates a list with the desired length, yet all entries equal to `None`. We maintain an additional instance variable, `_n`, that represents the number of actual entries currently in our table. For convenience, our class supports the `getitem` method to retrieve an entry at a given index with a syntax `board[i]` (or `None` if no such entry exists), and we support a simple `__str__` method that returns a string representation of the entire scoreboard, with one entry per line.

* (使用以上命令) 创建具有所需长度的列表，所有条目都为空。我们还设置了一个额外的实例变量 `_n`，用以表示表中当前的实际条目数。为了方便起见，我们的类支持 `__getitem__` 方法，可以使用语法 `board[i]` 返回给定下标 `i` 的对应值，并且我们支持一个简单的 `__str__` 方法，它返回整个记分板的字符串表示，每行一个条目。

Adding an Entry

* 添加条目

The most interesting method of the Scoreboard class is `add`, which is responsible for considering the addition of a new entry to the scoreboard. Keep in mind that every entry will not necessarily qualify as a high score. If the board is not yet full, any new entry will be retained. Once the board is full, a new entry is only retained if it is strictly better than one of the other scores, in particular, the last entry of the scoreboard, which is the lowest of the high scores.

*

When a new score is considered, we begin by determining whether it qualifies as a high score. If so, we increase the count of active scores, `n`, unless the board is already at full capacity. In that case, adding a new high score causes some other entry to be dropped from the scoreboard, so the overall number of entries remains the same.

*

To correctly place a new entry within the list, the final task is to shift any inferior scores one spot lower (with the least score being dropped entirely when the scoreboard is full). This process is quite similar to the implementation of the `insert` method of the list class, as described on pages 204–205. In the context of our scoreboard, there is no need to shift any `None` references that remain near the end of the array, so the process can proceed as diagrammed in Figure 5.19.

*

To implement the final stage, we begin by considering index `j = self.n - 1`, which is the index at which the last `GameEntry` instance will reside, after completing the operation. Either `j` is the correct index for the newest entry, or one or more immediately before it will have lesser scores. The while loop at line 34 checks the compound condition, shifting references rightward and decrementing `j`, as long as there is another entry at index `j - 1` with a score less than the new score.

*

5.5.2 Sorting a Sequence

* 序列排序

5.5.3 Simple Cryptography

* 5.5.3 节 简单密码学

An interesting application of strings and lists is **cryptography**, the science of secret messages and their applications. This field studies ways of performing **encryption**, which takes a message, called the **plaintext**, and converts it into a scrambled message, called the **ciphertext**. Likewise, cryptography also studies corresponding ways of performing **decryption**, which takes a ciphertext and turns it back into its original plaintext.

* 字符串和列表的一个有趣应用是密码学，即有关秘密消息的科学及其应用。该领域研究执行加密的方式，它把明文的消息，转换成为密文。同样，密码学也研究了执行解密的相应方式，它将密文转回原来的明文。

Arguably the earliest encryption scheme is the **Caesar cipher**, which is named after Julius Caesar, who used this scheme to protect important military messages. (All of Caesar's messages were written in Latin, of course, which already makes them unreadable for most of us!) The Caesar cipher is a simple way to obscure a message written in a language that forms words with an alphabet.

* 可以说，最早的加密方案是凯撒密码，凯撒密码是以朱利叶斯·凯撒命名的，他用这个方案来保护重要的军事信息。（凯撒的所有消息都是用拉丁文写的，当然这已经使我们大部分人都读不出来了！）凯撒密码是一种简单的方法，可以用来加密字母消息。

The Caesar cipher involves replacing each letter in a message with the letter that is a certain number of letters after it in the alphabet. So, in an English message, we might replace each A with D, each B with E, each C with F, and so on, if shifting by three characters. We continue this approach all the way up to W, which is replaced with Z. Then, we let the substitution pattern **wrap around**, so that we replace X with A, Y with B, and Z with C.

* 凯撒密码包括用字母表中的一定数量的字母替换消息中的每个字母。所以，在一个英文消息中，我们可以让 A 用 D 替换，B 用 E 替换，C 用 F 替换，依此类推。我们继续这种方式一直到 W，它被 Z 替换。然后，所以 X 用 A 替换，Y 用 B 替换，Z 用 C 替换。

Converting Between Strings and Character Lists

* 字符串和字符列表之间的转换

Given that strings are immutable, we cannot directly edit an instance to encrypt it. Instead, our goal will be to generate a new string. A convenient technique for performing string transformations is to create an equivalent list of characters, edit the list, and then reassemble a (new) string based on the list. The first step can be performed by sending the string as a parameter to the constructor of the list class. For example, the expression `list('bird')` produces the result `['b', 'i', 'r', 'd']`. Conversely, we can use a list of characters to build a string by invoking the `join` method on an empty string, with the list of

characters as the parameter. For example, the call `".join(['b', 'i', 'r', 'd'])` returns the string `'bird'`.

* 鉴于这些字符串是不可变的，我们无法直接编辑一个实例进行加密。相反，我们的目标是生成一个新的字符串。用于字符串转换的方法是创建一个与之等效的字符列表，然后编辑列表，根据列表重新组合一个字符串。可以通过将字符串作为参数发送到列表类来实现第一步。例如，表达式 `list('bird')` 生成结果 `['b', 'i', 'r', 'd']`。反之，我们可以使用字符列表来构建一个字符串，方法是在空字符串中引用 `join` 方法，其中以字符列表为参数。例如，`".join(['b', 'i', 'r', 'd'])` 返回字符串 `'bird'`。

Using Characters as Array Indices

* 把字符转化为数组下标

If we were to number our letters like array indices, so that A is 0, B is 1, C is 2, and so on, then we can write the Caesar cipher with a rotation of r as a simple formula: Replace each letter i with the letter $(i + r) \bmod 26$, where \bmod is the **modulo** operator, which returns the remainder after performing an integer division. This operator is denoted with `%` in Python, and it is exactly the operator we need to easily perform the wrap around at the end of the alphabet. For $26 \bmod 26$ is 0, $27 \bmod 26$ is 1, and $28 \bmod 26$ is 2. The decryption algorithm for the Caesar cipher is just the opposite—we replace each letter with the one r places before it, with wrap around (that is, letter i is replaced by letter $(i - r) \bmod 26$).

* 如果我们把数字编号为数组下标，所以 A 为 0，B 为 1，C 为 2，等等，那么我们可以用 r 的旋转写出凯撒密码：使用第 $(i + r) \bmod 26$ 个字母第 i 个字母（当然是字母表顺序），其中 \bmod 是模运算符，其在执行整数除法之后返回余数。这个操作符在 Python 中用 `%` 表示。对于 $26 \bmod 26$ 是 0， $27 \bmod 26$ 是 1， $28 \bmod 26$ 是 2。凯撒密码的解密算法恰恰相反，我们用一个 r 个地方替换每个字母，包围（即，第 i 个字母被替换为第 $(i - r) \bmod 26$ 个字母）。

We can represent a replacement rule using another string to describe the translation. As a concrete example, suppose we are using a Caesar cipher with a three-character rotation. We can precompute a string that represents the replacements that should be used for each character from A to Z. For example, A should be replaced by D, B replaced by E, and so on. The 26 replacement characters in order are DEFGHIJKLMN-OPQRSTUVWXYZABC. We can subsequently use this translation string as a guide to encrypt a message. The remaining challenge is how to quickly locate the replacement for each character of the original message.

* 我们可以使用另一个字符串来表示替换规则。作为一个具体的例子，假设我们使用一个三角形旋转的凯撒密码。我们可以预先计算一个字符串，代表每个字符从 A 到 Z 的加密。例如，A 应由 D 替换，B 替换为 E，依此类推。26 个重新排列字符顺序为 DEFGHIJKLMNOPQRSTUVWXYZABC。我们随后可以

使用这个字符串作为加密消息的指南。剩下的挑战是如何快速找到原始消息中每个字符的加密字符。

Fortunately, we can rely on the fact that characters are represented in Unicode by integer code points, and the code points for the uppercase letters of the Latin alphabet are consecutive (for simplicity, we restrict our encryption to uppercase letters). Python supports functions that convert between integer code points and one-character strings. Specifically, the function `ord(c)` takes a one-character string as a parameter and returns the integer code point for that character. Conversely, the function `chr(j)` takes an integer and returns its associated one-character string.

* 幸运的是，我们可以借助 Unicode 字符集，以整数代码点表示字符，拉丁文大写字母的 Unicode 值是连续的（为了简单起见，我们将加密限制为大写字母）。Python 支持在整数 Unicode 值和单字符之间转换的功能。具体来说，函数 `ord(c)` 采用单字符的字符串作为参数，并返回该字符的 Unicode 值。相反，函数 `chr(j)` 将整数 `j` 作为参数，返回一个与 `j` 相匹配的 Unicode 数值。

In order to find a replacement for a character in our Caesar cipher, we need to map the characters A to Z to the respective numbers 0 to 25. The formula for doing that conversion is $j = \text{ord}(c) - \text{ord}('A')$. As a sanity check, if character `c` is `A`, we have that $j = 0$. When `c` is `B`, we will find that its ordinal value is precisely one more than that for `A`, so their difference is 1. In general, the integer `j` that results from such a calculation can be used as an index into our precomputed translation string, as illustrated in Figure 5.21.

* 为了找到我们的凯撒密码中的字符替换，我们需要将字符 `A` 与 `Z` 映射到相应的数字 0 到 25。进行转换的公式是 $j = \text{ord}(c) - \text{ord}('A')$ 。为了稳健不妨检查一下，如果字符 `c` 是 `A`，我们有 $j = 0$ 。当 `c` 是 `B` 时，我们将发现它的

序数值比 `A` 的正数值多一个，所以它们的差是 1。一般来说，由此计算得到的 `j` 可以用作我们预先计算的转换字符串中的下标，如图 5.21 所示。

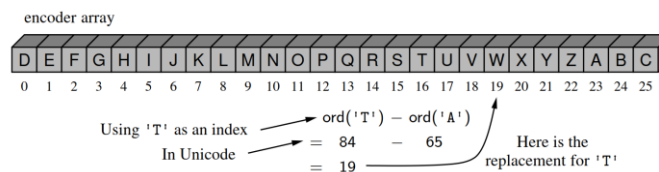


Figure 5.21: Illustrating the use of uppercase characters as indices, in this case to perform the replacement rule for Caesar cipher encryption.

In Code Fragment 5.11, we develop a Python class for performing the Caesar cipher with an arbitrary rotational shift, and demonstrate its use. When we run this program (to perform a simple test), we get the following output.

* 在代码片段 5.11 中，我们开发了一个 Python 类用于执行任意旋转移位的凯撒密码，并在后面演示了使用方法。当我们运行这个程序（执行一个简单的测试）时，我们得到以下输出：

```
Secret:  WKH HDJOH LV LQ SODB; PHHW DW MRH'V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S.
```

The constructor for the class builds the forward and backward translation strings for the given rotation. With those in hand, the encryption and decryption algorithms are essentially the same, and so we perform both by means of a nonpublic utility method named `_transform`.

* 类的构造函数为给定的位移量提供了向前和向后两种加密方式。加密和解密算法本质上是一样的，所以我们通过一个名为 `_transform` 的非 `public` 函数来执行这两种过程。

5.6 Multidimensional Data Sets

* 多维数据集

Lists, tuples, and strings in Python are one-dimensional. We use a single index to access each element of the sequence. Many computer applications involve multidimensional data sets. For example, computer graphics are often modeled in either two or three dimensions. Geographic information may be naturally represented in two dimensions, medical imaging may provide three-dimensional scans of a patient, and a company's valuation is often based upon a high number of independent financial measures that can be modeled as multidimensional data. A two-dimensional array is sometimes also called a *matrix*. We may use two indices, say i and j , to refer to the cells in the matrix. The first index usually refers to a row number and the second to a column number, and these are traditionally zero-indexed in computer science. Figure 5.22 illustrates a two-dimensional data set with integer values. This data might, for example, represent the number of stores in various regions of Manhattan.

* Python 中的列表, 元组和字符串是一维的。我们使用单个索引来访问序列的每个元素。许多计算机应用涉及多维数据集。例如, 计算机图形通常被建模在两个或三个维度。地理信息可以自然地代表在二维上, 医学成像可以提供患者的三维扫描, 并且公司报价通常可以被建模为多维数据。二维阵列有时也称为矩阵。我们可以使用两个指标, 比如 i 和 j 来指代矩阵中的单元格。第一个索引通常指的是行号, 而第二个索引通常指的是列数, 这些在计算机科学中通常是零索引的。图 5.22 说明了具有整数值的二维数据集 例如, 这些数据可能代表了曼哈顿各个地区的商店数量。

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figure 5.22: Illustration of a two-dimensional integer data set, which has 8 rows and 10 columns. The rows and columns are zero-indexed. If this data set were named `stores`, the value of `stores[3][5]` is 100 and the value of `stores[6][2]` is 632.

A common representation for a two-dimensional data set in Python is as a list of lists. In particular, we can represent a two-dimensional array as a list of rows, with each row itself being a list of values. For example, the two-dimensional data

* Python 中二维数据集的通用表示方式是列表的列表。特别地, 我们可以将二维数组作为行列表来表示, 每一行本身都是一个值列表。例如, 二维数据

```
22  18  709  5  33
45  32  830 120 750
4   880  45  66  61
```

might be stored in Python as follows.

* 在 Python 中可以存储为:

```
data = [[22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61]]
```

An advantage of this representation is that we can naturally use a syntax such as `data[1][3]` to represent the value that has row index 1 and column index 3, as `data[1]`, the second entry in the outer list, is itself a list, and thus indexable.

* 这种表示的一个优点是, 我们可以自然地使用诸如 `data[1][3]` 的语法来表示具有行索引 1 和列索引 3 的值作为 `data[1][3]`, 因为 `data[1]` 本身就是一个列表, 因此是可以通过下标访问。

To quickly initialize a one-dimensional list, we generally rely on a syntax such as `data = [0] * n` to create a list of n zeros. On page 189, we emphasized that from a technical perspective, this creates a list of length n with all entries referencing the same integer instance, but that there was no meaningful consequence of such aliasing because of the immutability of the `int` class in Python.

* 为了快速初始化一维列表, 我们通常依赖于诸如 `data = [0] * n` 的语法来创建一个含有 n 个零的列表。在第 189 页, 我们强调从技术的角度来看, 这将创建一个长度为 n 的列表, 所有条目引用相同的整数实例, 但是由于 Python 中 `int` 类的不可变性, 所以这 n 个 0 的别名没有意义。

We have to be considerably more careful when creating a list of lists. If our goal were to create the equivalent of a two-dimensional list of integers, with r rows and c columns, and to initialize all values to zero, a flawed approach might be to try the command

* 在创建列表的列表时, 我们必须非常小心。如果我们的目标是创建二维的 r 行 c 列整数列表, 并将所有值初始化为零, 可以尝试性地这样做:

```
data = ([0] * c) * r # Warning: this is a mistake
```

While `([0] * c)` is indeed a list of c zeros, multiplying that list by r unfortunately creates a single list with length $r \cdot c$, just as `[2,4,6] * 2` results in list `[2, 4, 6, 2, 4, 6]`.

* 虽然 `([0] * c)` 确实是一个含有 c 个 0 元素的列表, 但是将列表乘以 r 不幸地创建了一个长度为 $r \cdot c$ 的列表, 正如 `[2,4,6] * 2` 会生成 `[2, 4, 6, 2, 4, 6]`。

A better, yet still flawed attempt is to make a list that contains the list of c zeros as its only element, and then to multiply that list by r . That is, we could try the command

* 一个更好的但仍然是惊人的尝试是将列表中包含 c 个 0 的列表作为唯一的元素, 然后将该列表乘以 r 。也就是说, 我们可以尝试这个命令

```
data = [ [0] * c ] * r # Warning: still a mistake
```

This is much closer, as we actually do have a structure that is formally a list of lists. The problem is that all r entries of the

list known as `data` are references to the same instance of a list of `c` zeros. Figure 5.23 provides a portrayal of such aliasing.

* 这更接近，因为我们已经得到了一个存储列表的列表。问题是列表中的所有 r 个条目都是对同一个 c 列表列表的引用。图 5.23 提供了这种混叠的描述。

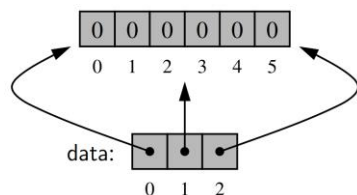


Figure 5.23: A flawed representation of a 3×6 data set as a list of lists, created with the command `data = [[0] * 6] * 3`. (For simplicity, we overlook the fact that the values in the secondary list are referential.)

This is truly a problem. Setting an entry such as `data[2][0] = 100` would change the first entry of the secondary list to reference a new value, 100. Yet that cell of the secondary list also represents the value `data[0][0]`, because “row” `data[0]` and “row” `data[2]` refer to the same secondary list.

* 这是一个真正的问题。如果改变了 `data[0][0]`，那么 `data` 这个一维数组的首元素也会随之改变，又因为 `data[1]` 和 `data[2]` 都指向引用了 `data`，所以也会跟着改变。这会导致混乱。（意译）

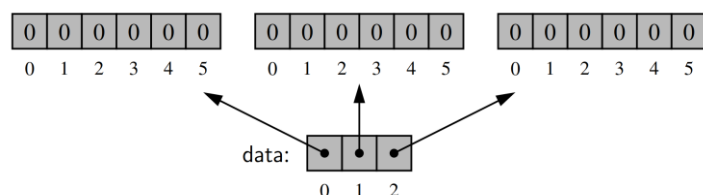


Figure 5.24: A valid representation of a 3×6 data set as a list of lists. (For simplicity, we overlook the fact that the values in the secondary lists are referential.)

To properly initialize a two-dimensional list, we must ensure that each cell of the primary list refers to an independent instance of a secondary list. This can be accomplished through the use of Python’s list comprehension syntax.

* 为了正确初始化二维列表，我们必须确保主列表的每个单元格引用独立实例。这可以通过使用 Python 的列表解释语法来完成。

```
data = [[0] * c for j in range(r)]
```

This command produces a valid configuration, similar to the one shown in Figure 5.24. By using list comprehension, the expression `[0] * c` is reevaluated for each pass of the embedded for loop. Therefore, we get r distinct secondary lists, as desired. (We note that the variable `j` in that command is irrelevant; we simply need a for loop that iterates r times.)

* 此命令产生一个有效二维数组，类似于图 5.24 所示。通过使用列表解析，使用了 for 循环，系统对 `[0] * c` 表达式进行重新评估。因此，根据需要，我们得到了不同的

里层数组。（我们注意到该命令中的变量 `j` 是无关紧要的；我们只需要一个循环遍历 r 次）。

Two-Dimensional Arrays and Positional Games

* 二维数组和位置游戏

Many computer games, be they strategy games, simulation games, or first-person conflict games, involve objects that reside in a two-dimensional space. Software for such positional games need a way of representing such a two-dimensional “board,” and in Python the list of lists is a natural choice.

* 许多电脑游戏，无论是战略游戏，模拟游戏还是第一人称冲突游戏，都涉及驻留在二维空间中的对象。用于这种位置游戏的软件需要一种代表这样的二维面板的方式，而在 Python 中，列表存储列表（即多维数组）是一个自然的选择。

Tic-Tac-Toe

* 井字棋

As most school children know, Tic-Tac-Toe is a game played in a three-by-three board. Two players—X and O—alternate in placing their respective marks in the cells of this board, starting with player X. If either player succeeds in getting three of his or her marks in a row, column, or diagonal, then that player wins.

* 大多数学校的孩子都知道，井字棋是一个三乘三网格的游戏。两名玩家 X 和 O 交替将其各自的标记放置在该棋盘的单元格中，从玩家 X 开始。如果任一玩家成功地在 一行，一列或对角线上的所有位置上放置了他的标记，则该玩家赢。

This is admittedly not a sophisticated positional game, and it’s not even that much fun to play, since a good player O can always force a tie. Tic-Tac-Toe’s saving grace is that it is a nice, simple example showing how two-dimensional arrays can be used for positional games. Software for more sophisticated positional games, such as checkers, chess, or the popular simulation games, are all based on the same approach we illustrate here for using a two-dimensional array for Tic-Tac-Toe.

* 这绝对不是一个复杂的定位游戏，玩起来甚至没有什么好玩，因为一个好的玩家 O 总是可以通过后出手，堵住对方而赢得比赛。Tic-Tac-Toe 的存在意义是，它可以作为一个很好的简单示例，显示二维数组如何用于定位游戏。更复杂的定位游戏的软件，例如棋子，棋牌或流行的模拟游戏，都是基于我们在这里使用 Tic-Tac-Toe 使用二维阵列的相同方法。

Our representation of a 3×3 board will be a list of lists of characters, with X or O designating a player’s move, or designating an empty space. For example, the board configuration

* 我们的 3 * 3 板的表示将是一个字符列表的列表，通过字符 X 或 O 标记玩家的举动，或指定一个空的位置。例如：

O	X	O
	X	
	O	X

will be stored internally as

*

`['O', 'X', 'O'], ['X', ' ', ' '], [' ', 'O', 'X']`

We develop a complete Python class for maintaining a Tic-Tac-Toe board for two players. That class will keep track of the moves and report a winner, but it does not perform any strategy or allow someone to play Tic-Tac-Toe against the computer. The details of such a program are beyond the scope of this chapter, but it might nonetheless make a good course project (see Exercise P-8.68).

* 我们开发了一个完整的 Python 类，用于为两名玩家记录 Tic-Tac-Toe 游戏过程。该 class 将跟踪动作并报告获胜者，但并不表示任何策略，也不能实现人机对战。这样一个程序（人机对战）的细节超出了本章的范围，但仍然可以做一个很好的课程（见练习 P-8.68）。

Before presenting the implementation of the class, we demonstrate its public interface with a simple test in Code Fragment 5.12.

* 在介绍课程的实现之前，我们通过 Code Code 5.12 中的简单测试来演示其公共接口。

The basic operations are that a new game instance represents an empty board, that the mark(i,j) method adds a mark at the given position for the current player (with the software managing the alternating of turns), and that the game board can be printed and the winner determined. The complete source code for the TicTacToe class is given in Code Fragment 5.13. Our mark method performs error checking to make sure that valid indices are sent, that the position is not already occupied, and that no further moves are made after someone wins the game.

* 基本操作是，新的游戏实例代表一个空板，mark(i,j)方法在当前玩家的给定位置添加一个标记（软件会进行出棋交替），并且游戏记录可以确定并打印获胜者。代码片段 5.13 中给出了 TicTacToe 类的完整源代码。我们的 mark 方法执行错误检查，以确保有效的索引被发送，而且发送的位置尚未被占用，并且在游戏结束之后没有进一步的移动。

END

六、实验体会

这次的实验报告相对比较简单，但是要从内存上面准确理解列表的作用原理，是 `Python` 编程的必备基础。所以我把第五章的课本大致翻译了一下。

最核心的东西，都在课本上说得很清楚，已经翻译过，这里就不再赘述。`string` 类的那么多个函数，很值得记忆。

七、参考文献

- [1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Python
- [2] 数据结构与算法分析：C 语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社
- [3] 算法导论（原书第三版），（美）科尔曼（Cormen, T.H.）等；殷建平等译. 北京：机械工业出版社