

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：2017-04-10
上机实践编号：No.06	组号：	上机实践时间：上午 3、4 节

一、实验目的

1. 熟悉与栈、队列等有关的数据结构与算法；
2. 熟悉主讲教材 Chapter 6 的代码片段。

二、实验内容

1. 线性表有关的数据结构设计及算法设计；
2. 调试主讲教材 Chapter 6 的 Python 程序；
3. 阅读实验教材第 2 章的问题，将 C 程序转化为 Python 程序（选做）。

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。

四、实验记录与实验结果分析

1 题

基于适配器模式，用数组实现栈逻辑结构

程序代码：

```
1  # 6.1.2 Simple Array-Based Stack Implementation
2
3  class Empty(Exception):
4      """Error attempting to access an element from an empty container."""
5      pass
6
7
8  class ArrayStack:
9      """LIFO Stack implementation using a Python list as underlying storage."""
10
11     def __init__(self):
12         """Create an empty stack."""
13         self._data = []           # nonpublic list instance
14
15     def __len__(self):
16         """Return the number of elements in the stack."""
17         return len(self._data)
18
19     def is_empty(self):
20         """Return True if the stack is empty."""
21         return len(self._data) == 0
22
```

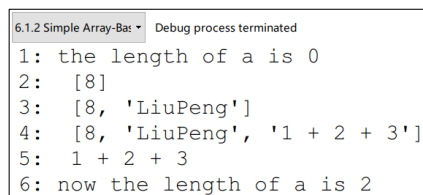
```

23     def push(self,e):
24         """Add element e to the top of the stack."""
25         self._data.append(e)           # new item stored at end of list
26
27     def top(self):
28         """Return (but do not remove) the element at the top of the stack.
29
30         Raise Empty exception if the stack is empty.
31         """
32         if self.is_empty():
33             raise Empty('Stack is empty')
34         return self._data[-1]           # the last item in the list
35
36     def pop(self):
37         """Remove and return the element from the top of the stack (i.e., LIFO).
38
39         Raise Empty exception if the stack is empty.
40         """
41         if self.is_empty():
42             raise Empty('Stack is empty')
43         return self._data.pop()         # remove last item from list
44
45 #----- my main function -----
46
47 a = ArrayStack()
48 print('1: the length of a is',a.__len__())
49 a.push(8)
50 print('2: ',a._data)
51 a.push('LiuPeng')
52 print('3: ',a._data)
53 a.push('1 + 2 + 3')
54 print('4: ',a._data)
55 print('5: ',a.pop())
56 print('6: now the length of a is',a.__len__())

```

程序代码 1

运行结果:



```

6.1.2 Simple Array-Bat  Debug process terminated
1: the length of a is 0
2:  [8]
3:  [8, 'LiuPeng']
4:  [8, 'LiuPeng', '1 + 2 + 3']
5:  1 + 2 + 3
6:  now the length of a is 2

```

运行结果 1

代码分析:

这几行代码，最核心的部分是用适配器模式，把一个列表作为底层实际结构，嵌入到类里面，使之看起来像是一个栈。这种模式是最终重要的，之前仅仅是草草看了这一段代码，翻译了一遍，还是有很多收获的。

## 2 题

利用栈逻辑结构，实现将一个文本文件（utf-8）按照行顺序反向输出。

程序代码：

```
1  # 6.1.3 Reversing Data Using a Stack
2
3  class Empty(Exception):
4      """Error attempting to access an element from an empty container."""
5      pass
6
7  class ArrayStack:
8      """LIFO Stack implementation using a Python list as underlying storage."""
9
10     def __init__(self):
11         """Create an empty stack."""
12         self._data = []           # nonpublic list instance
13
14     def __len__(self):
15         """Return the number of elements in the stack."""
16         return len(self._data)
17
18     def is_empty(self):
19         """Return True if the stack is empty."""
20         return len(self._data) == 0
21
22     def push(self, e):
23         """Add element e to the top of the stack."""
24         self._data.append(e)      # new item stored at end of list
25
26     def top(self):
27         """Return (but do not remove) the element at the top of the stack.
28
29         Raise Empty exception if the stack is empty.
30         """
31         if self.is_empty():
32             raise Empty('Stack is empty')
33         return self._data[-1]     # the last item in the list
34
35     def pop(self):
36         """Remove and return the element from the top of the stack (i.e., LIFO).
37
38         Raise Empty exception if the stack is empty.
39         """
40         if self.is_empty():
41             raise Empty('Stack is empty')
42         return self._data.pop()   # remove last item from list
43
44     def iter(self):
45         return iter(self._data)
```

```

46
47 import codecs
48 def reverse_file(filename):
49     """Overwrite given file with its contents line-by-line reversed."""
50     S = ArrayStack()
51     original = codecs.open(filename, 'r', encoding='utf-8')
52     for line in original:
53         S.push(line.rstrip('\n'))
54         # we will re-insert newlines whth writing
55     original.close
56
57     # now we ovrewrite with contents in LIFO order
58     output = codecs.open(filename, 'w', encoding='utf-8')
59     # reopening file overwrites original
60     while not S.is_empty():
61         output.write(S.pop() + '\n')    # re-insert newline characters
62     output.close()
63
64 #----- my main function -----
65
66 import codecs
67
68 S_1 = ArrayStack()
69 a = S_1.iter()
70 original = codecs.open('TEST_6.1.3.txt', 'r', encoding='utf-8')
71 for line in original:
72     S_1.push(line)
73 original.close
74 for i in a:
75     print(i.rstrip('\n'))
76
77 reverse_file('TEST_6.1.3.txt')
78
79 print('\n-----')
80 S_2 = ArrayStack()
81 a = S_2.iter()
82 original = codecs.open('TEST_6.1.3.txt', 'r', encoding='utf-8')
83 for line in original:
84     S_2.push(line)
85     # we will re-insert newlines whth writing
86 original.close
87 for i in a:
88     print(i.rstrip('\n'))

```

程序代码 2

运行结果:

```
6.1.3 Reversing Data L ▾ Debug I/O (stdin, stdout, stderr) appears below
我能吞下玻璃而不伤身体
-----
体身
伤不而璃玻下吞能我
```

运行结果 2

代码分析：

可以看到，这段代码与课本上的类稍有不同，加了一个 `iter` 方法，进行了迭代器生成。因为这里的 `ArrayStack` 就是用一个列表伪装了一个栈，所以内部必然支持迭代器生成。我加了一个迭代器，目的是输出所有的栈元素。当然，也可以通过文件，直接输出所有的行。

**3 题**

括号匹配与标记匹配。。

程序代码:

```

1  # 6.1.4 Matching Parentheses and HTML Tags
2
3  # 6.1.4.1 An Algorithm for Matching delimiters
4  # 6.1.4.2 Matching Tags in a Markup Language
5
6  class Empty(Exception):
7      """Error attempting to access an element from an empty container."""
8      pass
9
10
11 class ArrayStack:
12     """LIFO Stack implementation using a Python list as underlying storage."""
13
14     def __init__(self):
15         """Create an empty stack."""
16         self._data = []                # nonpublic list instance
17
18     def __len__(self):
19         """Return the number of elements in the stack."""
20         return len(self._data)
21
22     def is_empty(self):
23         """Return True if the stack is empty."""
24         return len(self._data) == 0
25
26     def push(self, e):
27         """Add element e to the top of the stack."""
28         self._data.append(e)           # new item stored at end of list
29
30     def top(self):
31         """Return (but do not remove) the element at the top of the stack.
32
33         Raise Empty exception if the stack is empty.
34         """
35         if self.is_empty():
36             raise Empty('Stack is empty')
37         return self._data[-1]          # the last item in the list
38
39     def pop(self):
40         """Remove and return the element from the top of the stack (i.e., LIFO).
41
42         Raise Empty exception if the stack is empty.
43         """
44         if self.is_empty():
45             raise Empty('Stack is empty')
46         return self._data.pop()        # remove last item from list

```

```

47
48 def is_amtched_html(raw):
49     """Return True if all HTML tags are properly match; False otherwise."""
50     S = ArrayStack()
51     j = raw.find('<>')          # find first '<' character (if any)
52     while j != -1:
53         k = raw.find('>',j+1)    # find next '>' character
54         if k ==1:
55             return False        # invalid tag
56         tag = raw[j+1:k]        # strip away < >
57         if not tag.startswith('/'): # this is opening tag
58             S.push(tag)
59         else:                    # this is closing tag
60             if S.is_empty():
61                 return False    # nothing to match with
62             if tag[1:] != S.pop():
63                 return False    # mismatched delimiter
64             j = raw.find('<',k+1)  # find next '<' character (if any)
65     return S.is_empty()         # were all opening tags matched?
66
67 def is_matched(expr):
68     """Return True if all delimiters are properly match; False otherwise."""
69     lefty = '({['              # opening delimiters
70     righty = ')}]'             # respective closing delims
71     S = ArrayStack()
72     for c in expr:
73         if c in lefty:
74             S.push(c)           # push left delimiter on stack
75         elif c in righty:
76             if S.is_empty():
77                 return False    # nothing to match with
78             if righty.index(c) != lefty.index(S.pop()):
79                 return False    # mismatched
80     return S.is_empty()         # were all symbols matched
81
82 #----- my main function -----
83 test = '{[(1 + x) * c - (d - e)] / m} + q'
84 print(is_matched(test))

```

程序代码 3

运行结果:

6.1.4 Matching Parent    Debug I/O (stdin, stdout, stderr) appears below  
True

运行结果 3

代码分析:

这段代码把两个匹配程序都写在了一个文件里。首先看 `is_matched` 函数, 可以看到, 遇到左括号就压栈, 如果遇到右括号, 就进行匹配检查。因为我们知道一个前期现象, 就是三种括号是由实际优先级的, 所以, 一批左括号之后, 就是优

优先级最小的右括号。根据这个原理，可以知道两个左右括号之间如果没有括号，那么这两个括号必然是匹配的。



## 4 题

队列的 Python 实现，用适配器，基于动态数组。

程序代码：

```

1  # 6.2.2 Array-Based Queue Implementation
2
3  class Empty(Exception):
4      """Error attempting to access an element from an empty container."""
5      pass
6
7  class ArrayQueue:
8      """FIFO queue implementation using a Python list as underlying storage."""
9      DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
10
11     def __init__(self):
12         """Create an empty queue."""
13         self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
14         self._size = 0
15         self._front = 0
16
17     def __len__(self):
18         """Return the number of element in the queue."""
19         return self._size
20
21     def is_empty(self):
22         """Return True if the queue if empty."""
23         return self._size == 0
24
25     def first(self):
26         """Return (but do not remove) the element at the front of the queue.
27
28         Raise Empty exception if the queue is empty.
29         """
30         if self.is_empty():
31             raise Empty('Queue is empty')
32         return self._data[self._front]
33
34     def dequeue(self):
35         """Remove and return the first element of the queue (i.e., FIFO).
36
37         Raise Empty exception if the queue is empty.
38         """
39         if self.is_empty():
40             raise Empty('Queue is empty')
41         answer = self._data[self._front]
42         self._data[self._front] = None        # help garbage collection
43         self._front = (self._front + 1) % len(self._data)
44         self._size -= 1
45         return answer

```

```

46
47     def enqueue(self,e):
48         """Add an element to the back of queue."""
49         if self._size == len(self._data):
50             self._resize(2 * len(self._data))# double the array size
51         avail = (self._front + self._size) % len(self._data)
52         self._data[avail] = e
53         self._size += 1
54
55     def _resize(self,cap):                # we assume cap >= len(self)
56         """Resize to a new list of capacity >= len(self)."""
57         old = self._data                  # keep track of existing list
58         self._data = [None] * cap         # allocate list with new capacity
59         walk = self._front
60         for k in range(self._size):       # only consider existing elements
61             self._data[k] = old[walk]     # intentionally shift indices
62             walk = (1 + walk) % len(old)   # use old size as modules
63         self._front = 0                   # front has been realigned
64
65 #----- my main function -----
66
67 a = ArrayQueue()
68 print('1: ',a.__len__(),' ,front is',a._front)
69 a.enqueue('LiuPeng')
70 print('2: ',a._data,' , front is',a._front)
71 for i in range(9):
72     a.enqueue(i)
73 print('3: ',a._data,' ,front is',a._front)
74 a.dequeue()
75 print('4: ',a._data,' , front is',a._front)
76 a.dequeue()
77 print('5: ',a._data,' , front is',a._front)
78 for i in range(10):
79     a.enqueue(i * i)
80 print('6: ',a._data,' , front is',a._front)

```

程序代码 4

运行结果:

```

6.2.2 Array-Based Queue Debug I/O (stdin, stdout, stderr) appears below
1:  0 ,front is 0
2:  ['LiuPeng', None, None, None, None, None, None, None, None, None] , front is 0
3:  ['LiuPeng', 0, 1, 2, 3, 4, 5, 6, 7, 8] ,front is 0
4:  [None, 0, 1, 2, 3, 4, 5, 6, 7, 8] , front is 1
5:  [None, None, 1, 2, 3, 4, 5, 6, 7, 8] , front is 2
6:  [1, 2, 3, 4, 5, 6, 7, 8, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, None, None] , front is 0

```

运行结果 4

代码分析:

代码比较简单。最核心的部分，在学习完适配器模式之后，已经不再神秘。但是能想出用循环使用一个线性表来实现内存的高效利用还是非常机智的。为了实现循环，这里引入了模运算。当  $a \bmod b$  中的  $a$  小于  $b$  时，就返回  $a$ ，这可以用来

进行 `front` 的前进;  $(\text{front} + \text{size}) \bmod \text{len}(\text{data})$  实现了入队的位置确定。这能实现, 全仗模运算的有效利用。因为模运算本身具有周期性。

而 `resize` 函数, 在扩充的同时, 把数组理了一遍, 把 `None` 全都放在了后面。这样一来可以避免很多问题, 还是拿 figure 6.7 看, 这是很显然的, 如果数组满了, 然而 `front` 仍然不变, 那么进行 `enqueue` 时,  $(\text{front} + \text{size}) \bmod \text{len}(\text{data}) = \text{front} + \text{size} \neq \text{size}$ , 这时候就会使一部分区域被跨越了! 然后就会有紊乱的情况发生, 申请不再明确。

五、教材翻译

Translation

Chapter 6 Stacks, Queues, and Deques

\*第六章 栈，队列，双端队列

6.1 Stacks

\*6.1 节 栈

A *stack* is a collection of objects that are inserted and removed according to the *last-in, first-out (LIFO)* principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack). The name “stack” is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the “pushing” and “popping” of plates on the stack. When we need a new plate from the dispenser, we “pop” the top plate off the stack, and when we add a plate, we “push” it down on the stack to become the new top plate. Perhaps an even more amusing example is a PEZ® candy dispenser, which stores mint candies in a spring-loaded container that “pops” out the topmost candy in the stack when the top of the dispenser is lifted (see Figure 6.1). Stacks are a fundamental data structure. They are used in many applications, including the following.

\* 堆栈是根据后进先出（LIFO）原则插入和重新移动的对象集合。用户可以随时将对象插入堆栈，但只能访问或删除最近插入的对象（在堆栈的“顶部”）。stack（栈）这个词源于弹簧加载的自助餐盘分配器中的一叠板的比喻。在这种情况下，基本操作涉及堆叠板上的“推”和“弹”。当我们从分配器取出一个盘子时，我们将顶板从堆叠中“弹出（pop）”，当我们添加盘子时，我们将其推下（push）。一个更加有趣的例子是 PEZ®糖果分配器，它将薄荷糖储存在一个弹簧加载的容器中，当分配器的顶部被提起时，它将“弹”出堆叠中最高的糖果（见图 6.1）。堆栈是基础数据结构。它们用于许多应用，包括以下内容。

**Example 6.1:** Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site’s address is “pushed” onto the stack of addresses. The browser then allows the user to “pop” back to previously visited sites using the “back” button.

\* 例 6.1: Internet Web 浏览器将最近访问的站点的地址存储在堆栈中。每次用户访问新站点时，该站点的地址被“压”到堆栈里。然后，浏览器允许用户使用“返回”按钮“弹出”回到以前访问过的网站。

**Example 6.2:** Text editors usually provide an “undo” mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.

\* 例 6.2: 文本编辑器通常提供一种“撤消”机制，可以取消最近的编辑操作并恢复到文档的前一个状态。这种撤消

操作可以通过在堆栈中保持文本更改来完成。

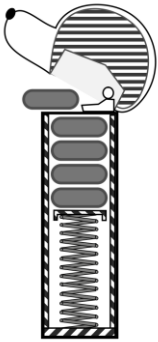


Figure 6.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

6.1.1The Stack Abstract Data Type

\* 6.1.1 节 栈的抽象数据结构

Stacks are the simplest of all data structures, yet they are also among the most important. They are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms. Formally, a stack is an abstract data type (ADT) such that an instance *S* supports the following two methods:

\* 堆栈是所有数据结构中最简单的，但它也是最重要的。它们被用于许多不同的应用程序，并且作为许多更复杂的数据结构和算法的工具。正式来看，堆栈是抽象数据类型（ADT），一个堆栈 *S* 支持以下两种方法：

- S.push(e):** Add element *e* to the top of stack *S*.  
\* 把元素 *e* 压入栈 *S* 内
- S.pop():** Remove and return the top element from the stack *S*; an error occurs if the stack is empty.  
\* 把栈顶部的元素删除同时返回这个元素的数值，如果栈是空的，就会抛出异常。

Additionally, let us define the following accessor methods for convenience:

\* 另外，为方便起见，我们定义以下访问方法：

- S.top:** Return a reference to the top element of stack *S*, without removing it; an error occurs if the stack is empty.  
\* 返回堆栈 *S* 的顶部元素，而不删除它；如果堆栈为空，则会抛出异常。
- S.is empty():** Return True if stack *S* does not contain any elements.  
\* 当堆栈为空的时候，返回 True。
- len(S):** Return the number of elements in stack *S*; in Python, we implement this with the special method `__len__`.  
\* 返回栈 *S* 中的元素数；在 Python 中，我们使用名为 `__len__` 的特殊方法来实现。

By convention, we assume that a newly created stack is empty, and that there is no a priori bound on the capacity of the stack. Elements added to the stack can have arbitrary type.

\* 按照惯例，我们假设一个新创建的堆栈是空的，并且没有对堆栈容量的前期约束。添加到堆栈的元素可以是任意类型。

**Example 6.3:** The following table shows a series of stack operations and their effects on an initially empty stack *S* of integers.

\* 下表显示了一系列栈操作及其对最初空栈 *S* 的影响。

Operation	Return Value	Stack Contents
S.push(5)	—	[5]
S.push(3)	—	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]
S.is_empty()	True	[ ]
S.pop()	“error”	[ ]
S.push(7)	—	[7]
S.push(9)	—	[7, 9]
S.top()	9	[7, 9]
S.push(4)	—	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	—	[7, 9, 6]
S.push(8)	—	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]

6.1.2 Simple Array-Based Stack Implementation

★ 基于数组简单的堆栈实现

We can implement a stack quite easily by storing its elements in a Python list. The list class already supports adding an element to the end with the append method, and removing the last element with the pop method, so it is natural to align the top of the stack at the end of the list, as shown in Figure 6.2.

★ 我们可以通过将其元素存储在 Python 列表中来轻松实现栈。列表类已经支持使用 append 方法向元素添加元素，并使用 pop 方法删除最后一个元素，因此把列表的末尾元素设置为栈的顶部是很自然的，如图 6.2。



Figure 6.2: Implementing a stack with a Python list, storing the top element in the rightmost cell.

Although a programmer could directly use the list class in place of a formal stack class, lists also include behaviors (e.g., adding or removing elements from arbitrary positions) that would break the abstraction that the stack ADT represents. Also, the terminology used by the list class does not precisely align with traditional nomenclature for a stack ADT, in particular the distinction between append and push. Instead, we demonstrate how to use a list for internal storage while providing a public interface consistent with a stack.

★ 尽管程序员可以直接使用列表类代替正式的堆栈类，但列表可能会破坏栈 ADT 表示的抽象的行为（例如，添加或删除任意位置的元素）。此外，列表类使用的术语与传统的堆栈 ADT 的术语不完全一致，特别是 append 和 push。相反，我们演示如何使用列表进行内部存储，同时提供与堆栈一致的公共接口。

The Adapter Pattern

★ 适配器模式

The *adapter* design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface. One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable. By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class, but repackaged in a more convenient way. In the context of the stack ADT, we can adapt Python’s list class using the correspondences shown in Table 6.1.

★ 适配器设计模式适用于我们有效地修改现有类，使这个类可以与和它相关但不同的类相匹配。应用适配器模式的一般方法是定义一个新类，使其隐含现有类，然后使用该隐藏实例的方法来实现新类的每个方法。通过这种方式实现适配器模式，我们创建了一个新类，它可以执行与现有

类相同的功能，但以更方便的方式重新打包。在栈 ADT 的上下文中，我们可以使用表 6.1 所示的对应关系来调整 Python 的列表类。

Stack Method	Realization with Python list
S.push(e)	L.append(e)
S.pop()	L.pop()
S.top()	L[-1]
S.is_empty()	len(L) == 0
len(S)	len(L)

Table 6.1: Realization of a stack S as an adaptation of a Python list L.

Implementing a Stack Using a Python List

★ 使用 Python 列表实现栈结构

We use the adapter design pattern to define an ArrayStack class that uses an underlying Python list for storage. (We choose the name ArrayStack to emphasize that the underlying storage is inherently array based.) One question that remains is what our code should do if a user calls pop or top when the stack is empty. Our ADT suggests that an error occurs, but we must decide what type of error. When pop is called on an empty Python list, it formally raises an IndexError, as lists are index-based sequences. That choice does not seem appropriate for a stack, since there is no assumption of indices. Instead, we can define a new exception class that is more appropriate. Code Fragment 6.1 defines such an Empty class as a trivial subclass of the Python Exception class.

★ 我们使用适配器设计模式来定义使用底层 Python 列表进行存储的 ArrayStack 类。（我们选择名称 ArrayStack 来强调存储本质上是基于数组的。）遗留的一个问题是当堆栈为空时用户调用 pop 或 top 时，我们的代码应该做什么。我们的 ADT 建议抛出异常，但是我们必须决定抛出什么类型的错误。当在空的 Python 列表上调用 pop 时，它会引发名为 IndexError 的错误，因为列表是基于索引的序列。这个名称似乎不适合栈，因为栈没有下标索引。相反，我们可以定义一个更适合的新的异常类。代码片段 6.1 将这样一个空类定义为 Python Exception 类的一个简单的子类。

```
class Empty(Exception):
    """Error attempting to access an element
    from an empty container."""
    pass
```

The formal definition for our ArrayStack class is given in Code Fragment 6.2. The constructor establishes the member self.data as an initially empty Python list, for internal storage. The rest of the public stack behaviors are implemented, using the corresponding adaptation that was outlined in Table 6.1.

★ 我们的 ArrayStack 类的形式定义在 Code Fragment 6.2。构造函数建立列表类。data 作为初始空白的 Python 列表，用于内部存储。公共栈行为的其余部分使用表 6.1 中概述的相应的适配器来实现。

Example Usage

★ 使用示例

Below, we present an example of the use of our ArrayStack class, mirroring the operations at the beginning of Example 6.3 on page 230.  
\* 对照着第 230 页的示例 6.3 开头的操作，我们举一个使用 ArrayStack 类的例子，

S = ArrayStack( )	# contents: [ ]	
S.push(5)	# contents: [5]	
S.push(3)	# contents: [5, 3]	
print(len(S))	# contents: [5, 3];	outputs 2
print(S.pop())	# contents: [5];	outputs 3
print(S.is_empty())	# contents: [5];	outputs False
print(S.pop())	# contents: [ ];	outputs 5
print(S.is_empty())	# contents: [ ];	outputs True
S.push(7)	# contents: [7]	
S.push(9)	# contents: [7, 9]	
print(S.top())	# contents: [7, 9];	outputs 9
S.push(4)	# contents: [7, 9, 4]	
print(len(S))	# contents: [7, 9, 4];	outputs 3
print(S.pop())	# contents: [7, 9];	outputs 4
S.push(6)	# contents: [7, 9, 6]	



### 6.1.3 Revising Data Using a Stack

#### \* 6.1.3 节 利用栈结构反转数据

As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence. For example, if the values 1, 2, and 3 are pushed onto a stack in that order, they will be popped from the stack in the order 3, 2, and then 1.

\* 有了 LIFO 协议，可以将栈用做反转数据序列的通用工具。例如，如果值 1，2 和 3 按照顺序被推到堆栈上，则它们将按照 3，2，1 的顺序从栈中弹出。

This idea can be applied in a variety of settings. For example, we might wish to print lines of a file in reverse order in order to display a data set in decreasing order rather than increasing order. This can be accomplished by reading each line and pushing it onto a stack, and then writing the lines in the order they are popped. An implementation of such a process is given in Code Fragment 6.3.

\* 这个想法可以应用于各种设置。例如，我们可能希望把一个文件的所有行按照逆序打印出来，以便以递减的顺序显示一个数据集。这可以通过读取每行并将其压到栈里，然后按照它们弹出的顺序来输出。代码 6.3 中给出了这样一个过程的实现。

One technical detail worth noting is that we intentionally strip trailing newlines from lines as they are read, and then re-insert newlines after each line when writing the resulting file. Our reason for doing this is to handle a special case in which the original file does not have a trailing newline for the final line. If we exactly echoed the lines read from the file in reverse order, then the original last line would be followed (without newline) by the

original second-to-last line. In our implementation, we ensure that there will be a separating newline in the result.

\* 值得注意的一个细节是，我们在读取时有意从行中删除尾随的换行符，然后在写入结果文件后，在每之后重新插入换行符。我们这样做的原因是为了处理一个特殊情况，即原始文件没有最后一行的换行符。如果我们以相反的顺序完全输出文件读取的行，则原始的最后一行将按照原始的第二到最后一行进行跟踪（无换行）。在我们的实现中，我们确保在结果中将有一个分隔的换行符。

The idea of using a stack to reverse a data set can be applied to other types of sequences. For example, Exercise R-6.5 explores the use of a stack to provide yet another solution for reversing the contents of a Python list (a recursive solution for this goal was discussed in Section 4.4.1). A more challenging task is to reverse the order in which elements are stored within a stack. If we were to move them from one stack to another, they would be reversed, but if we were to then replace them into the original stack, they would be reversed again, thereby reverting to their original order. Exercise C-6.18 explores a solution for this task.

\* 使用栈来反转数据集的想法可以应用于其他类型的序列。例如，练习 R-6.5 探讨了使用栈来提供另一个解决 Python 列表内容的解决方案（第 4.4.1 节讨论了此目标的递归解决方案）。一个更具挑战性的任务是将栈中元素存储的顺序颠倒。如果我们将它们从一个堆栈移动到另一个堆栈，那么它们将被颠倒，但是如果我们将它们替换成原来的堆栈，那么它们将再次被反转，从而恢复原来的顺序。练习 C-6.18 探讨了此任务的解决方案。



#### 6.1.4 Matching Parentheses and HTML Tags

##### \* 括号和 HTML 标签的匹配

In this subsection, we explore two related applications of stacks, both of which involve testing for pairs of matching delimiters. In our first application, we consider arithmetic expressions that may contain various pairs of grouping symbols, such as

\* 在本小节中，我们将探讨堆栈的两个相关应用程序，这两个应用程序都涉及到匹配分隔符对。在我们的第一个应用程序中，我们考虑可能包含各种括号的算术表达式，例如：

- Parentheses: “(” and “)”  
\* 圆括号
- Braces: “{” and “}”  
\* 花括号
- Brackets: “[” and “]”  
\* 方括号

Each opening symbol must match its corresponding closing symbol. For example, a left bracket, “[,” must match a corresponding right bracket, “],” as in the expression  $[(5 + x) - (y + z)]$ . The following examples further illustrate this concept:

\* 每个开口符号必须与其相应的关闭符号相匹配。如表达式  $[(5 + x) - (y + z)]$  中的左括号 “[”，必须与对应的右括号 “]” 相匹配。以下示例进一步说明了这一概念：

- Correct:  $()() \{ ([()]) \}$   
\* 正确
- Correct:  $((()((() \{ ([()]) \})))$   
\* 正确
- Incorrect:  $)(() \{ ([()]) \}$   
\* 不正确
- Incorrect:  $([ [ ] )$   
\* 不正确
- Incorrect:  $($   
\* 不正确

We leave the precise definition of a matching group of symbols to Exercise R-6.6.

\* 我们将匹配的符号组的精确定义留给练习 R-6.6。

#### An Algorithm for Matching Delimiters

##### \* 匹配分隔符的算法

An important task when processing arithmetic expressions is to make sure their delimiting symbols match up correctly. Code Fragment 6.4 presents a Python implementation of such an algorithm. A discussion of the code follows.

\* 处理算术表达式的一个重要任务是确保它们的分界符号正确匹配。代码片段 6.4 提出了一种这样的算法的 Python 实现。以下是有关的讨论。

We assume the input is a sequence of characters, such as  $[(5+x)-(y+z)]$ . We perform a left-to-right scan of the original sequence, using a stack  $S$  to facilitate the matching of grouping symbols. Each time we encounter an opening symbol, we push that symbol onto  $S$ , and each time we encounter a closing symbol, we pop a symbol from the stack  $S$  (assuming  $S$  is not empty), and check that these two symbols form a valid pair. If we reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, there must be an opening delimiter on the stack without a matching symbol.

\* 我们假设输入是一系列字符，例如  $[(5 + x) - (y + z)]$ 。我们使用栈  $S$  来执行原始序列的从左到右的扫描，以便于分组符号的匹配。每次我们遇到一个开头符号，我们把这个符号压到  $S$  里，每次遇到一个关闭符号时，我们从堆栈  $S$  中弹出一个符号（假设  $S$  不为空），并且检查这两个符号是否形成一个有效的对。如果我们达到表达式的结尾，并且堆栈是空的，那么原始表达式被正确匹配。否则，栈里肯定有一个开头分隔符，而没有与之匹配的符号。

If the length of the original expression is  $n$ , the algorithm will make at most  $n$  calls to push and  $n$  calls to pop. Those calls run in a total of  $O(n)$  time, even considering the amortized nature of the  $O(1)$  time bound for those methods. Given that our selection of possible delimiters,  $\{[,$  has constant size, auxiliary tests such as  $c$  in `lefty` and `righty.index(c)` each run in  $O(1)$  time. Combining these operations, the matching algorithm on a sequence of length  $n$  runs in  $O(n)$  time.

\* 如果原始表达式的长度为  $n$ ，算法最多分别进行  $n$  次 push 和  $n$  次 pop 操作。即使考虑到这些方法的  $O(1)$  时间复杂度，这些调用总共运行在  $O(n)$  个时间内。考虑到我们选择可能的分隔符  $\{[,$  具有恒定大小，所以 `lefty` 和 `righty.index(c)` 语句都运行在  $O(1)$  时间以内。结合这些操作，匹配算法在长度  $n$  的输入下，将在  $O(n)$  时间内运行。

#### Matching Tags in a Markup Language

##### \* 标记语言中的匹配标签

Another application of matching delimiters is in the validation of markup languages such as HTML or XML. HTML is the standard format for hyperlinked documents on the Internet and XML is an extensible markup language used for a variety of structured data sets. We show a sample HTML document and a possible rendering in Figure 6.3.

\* 匹配分隔符的另一个应用是标记语言（如 HTML 或 XML）的验证。HTML 是互联网上超链接文档的标准格式，XML 是用于各种结构化数据集的可扩展标记语言。我们在图 6.3 中显示一个示例 HTML 文档和可能的渲染。

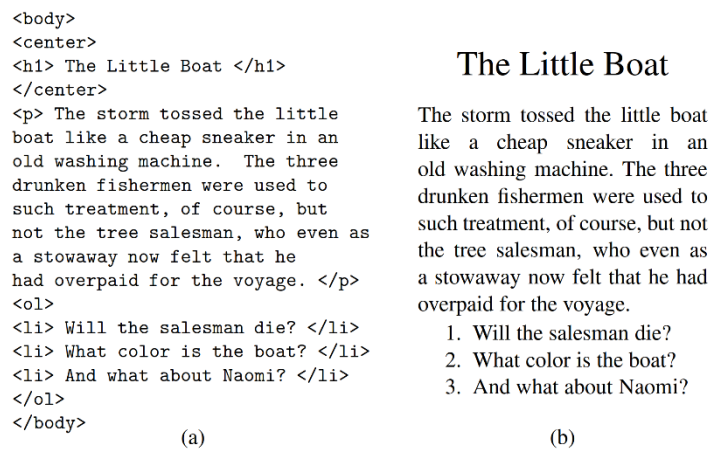


Figure 6.3: Illustrating HTML tags. (a) An HTML document; (b) its rendering.

In an HTML document, portions of text are delimited by **HTML tags**. A simple opening HTML tag has the form “<name>” and the corresponding closing tag has the form “</name>”. For example, we see the <body> tag on the first line of Figure 6.3(a), and the matching </body> tag at the close of that document. Other commonly used HTML tags that are used in this example include:

\* 在 HTML 文档中，部分文本由 HTML 标签分隔。一个简单的开放 HTML 标签的格式为“<name>”，相应的结束标签的格式为“</name>”。例如，我们看到图 6.3 (a) 的第一行的<body>标签，以及该文档附近匹配的</body>标签。本示例中使用的其他常用的 HTML 标签包括：

- body: document body  
\* body, 文件体
- h1: section header  
\* h1, 节开头

- center: center justify  
\* center, 居中对齐
- p: paragraph  
\* p, 段落
- ol: numbered (ordered) list  
\* ol, 有序表
- li: list item  
\* li, 表格

Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags. In Code Fragment 6.5, we give a Python function that matches tags in a string representing an HTML document. We make a left-to-right pass through the raw string, using index  $j$  to track our progress and the find method of the str class to locate the < and > characters that define the tags. Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack, just as we did when matching delimiters in Code Fragment 6.4. By similar analysis, this algorithm runs in  $O(n)$  time, where  $n$  is the number of characters in the raw HTML source.

\* 标签。在代码片段 6.5 中，我们给出一个 Python 函数，它匹配表示 HTML 文档的字符串中的标签。我们通过原始字符串进行从左到右的遍历，使用索引  $j$  跟踪我们的进度和 str 类的找到方法，以找到定义标签的<and>字符。打开标签被推到堆栈上，并且与堆栈中弹出的结束标签相匹配，就像我们在 6.4 中匹配分隔符时一样。通过类似的分析，该算法在  $O(n)$  时间运行，其中  $n$  是原始 HTML 源中的字符数。

## 6.2 Queue

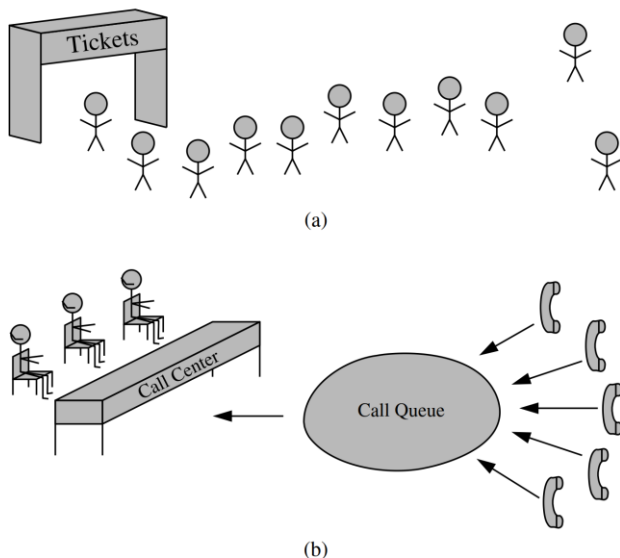
\*

Another fundamental data structure is the *queue*. It is a close “cousin” of the stack, as a queue is a collection of objects that are inserted and removed according to the *first-in, first-out (FIFO)* principle. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.

\* 另一个基本的数据结构就是队列。它是栈的“表亲”，因为队列是根据先进先出（FIFO）原则插入和删除的对象的集合。也就是说，元素可以随时被插入，但只有队列中最先进入的元素才能被删除。

We usually say that elements enter a queue at the back and are removed from the front. A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the front of the line. There are many other applications of queues (see Figure 6.4). Stores, theaters, reservation centers, and other similar services typically process customer requests according to the FIFO principle. A queue would therefore be a logical choice for a data structure to handle calls to a customer service center, or a wait-list at a restaurant. FIFO queues are also used by many computing devices, such as a networked printer, or a Web server responding to requests.

\* 我们通常会说，元素在后面进入队列，并从前面移除。打个比方，在公园里，有一队人等着娱乐自行车的使用权利，等待的人们在队伍的后面，从队伍前面开始排队。队列还有其他许多应用（参见图 6.4）。商店，剧院，预订中心和其他类似服务通常根据 FIFO 原则处理客户请求。因此，队列将成为处理对客户服务中心的呼叫或餐厅等待列表的逻辑选择。许多计算设备也使用 FIFO 队列，如网络打印机或响应请求的 Web 服务器。



**Figure 6.4:** Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.

### 6.2.1 The Queue Abstract Data Type

#### \* 队列数据结构

Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the *first* element in the queue, and element insertion is restricted to the back of the sequence. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle. The *queue* abstract data type (ADT) supports the following two fundamental methods for a queue Q:

\* 正式地，队列抽象数据类型定义了将对象保持在序列中的集合，其中元素访问和删除被限制在队列中的第一个元素，并且元素插入被限制在序列的后面。这种限制强制规则，按照先进先出（FIFO）原则，在队列中插入和删除项目。一个队列抽象数据类型（ADT）Q 支持以下两种基本方法：

**Q.enqueue(e):** Add element e to the back of queue Q.

\* 在队列 Q 的尾部添加一个元素。

**Q.dequeue():** Remove and return the first element from queue Q; an error occurs if the queue is empty.

\* 删除并返回队列 Q 的第一个元素，如果队列为空，就抛出异常。

The queue ADT also includes the following supporting methods (with first being analogous to the stack’s top method):

\* 队列 ADT 还包括以下支持方法（first 方法类似于栈的 top 方法）：

**Q.first():** Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.

\* 返回对队列 Q 前面的元素的引用，而不删除它；如果队列是空的，则会发生错误。

**Q.is\_empty():** Return True if queue Q does not contain any elements.

\* 如果队列 Q 不包含任何元素，则返回 True。

**len(Q):** Return the number of elements in queue Q; in Python, we implement this with the special method `__len__`.

\* 返回队列 Q 中的元素数；在 Python 中，我们使用 `__len__` 的特殊方法来实现。

By convention, we assume that a newly created queue is empty, and that there is no a priori bound on the capacity of the queue. Elements added to the queue can have arbitrary type.

\* 按照惯例，我们假设一个新创建的队列是空的，并且对队列的容量没有先验约束。添加到队列中的元素可以是任意类型。

**Example 6.4:** The following table shows a series of queue operations and their effects on an initially empty queue  $Q$  of integers.  
\* 示例 6.4: 下表显示了一系列队列操作及其对最初空队列整数  $Q$  的影响。

Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.is_empty()	False	[3]
Q.dequeue()	3	[ ]
Q.is_empty()	True	[ ]
Q.dequeue()	“error”	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first()	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue()	7	[9, 4]

### 6.2.2 Array-Based Queue Implementation

★

For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage. It may be very tempting to use a similar approach for supporting the queue ADT. We could enqueue element  $e$  by calling `append(e)` to add it to the end of the list. We could use the syntax `pop(0)`, as opposed to `pop()`, to intentionally remove the *first* element from the list when dequeuing.

★ 对于堆栈 ADT，我们创建了一个非常简单的适配器类，它使用 Python 列表作为底层存储。使用类似的方法支持队列 ADT 可能非常诱人。我们可以通过调用 `append(e)` 来将元素  $e$  添加到列表的末尾。我们可以使用语法 `pop(0)`，而不是 `pop()`，在出队时删除列表中的第一个元素。

As easy as this would be to implement, it is tragically inefficient. As we discussed in Section 5.4.1, when `pop` is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left, so as to fill the hole in the sequence caused by the `pop`. Therefore, a call to `pop(0)` always causes the worst-case behavior of  $\Theta(n)$  time.

★ 尽管采用这样的方法实现队列很容易，但它是不合理的。正如我们在 5.4.1 节中讨论的那样，当在具有非默认下标 (`pop()`) 的列表上调用 `pop` 时，程序将会执行一个循环，将所有在右边的元素向右搬移一个位置，以便下标索引。因此，对 `pop(0)` 的调用总是导致  $\Theta(n)$  最坏时间复杂度。

We can improve on the above strategy by avoiding the call to `pop(0)` entirely. We can replace the dequeued entry in the array with a reference to `None`, and maintain an explicit variable  $f$  to store the index of the element that is currently at the front of the queue. Such an algorithm for dequeue would run in  $O(1)$  time. After several dequeue operations, this approach might lead to the configuration portrayed in Figure 6.5.

★ 我们可以通过一些措施避免调用 `pop(0)` 的默认策略。我们可以把指向替换为 `None`，来改善出队操作，并通过一个变量  $f$  来存储当前在队列前面的元素。这种出队算法将在  $O(1)$  时间内运行。经过几次出队操作，这种方法可能会导致图 6.5 所示的状态。



Figure 6.5: Allowing the front of the queue to drift away from index 0.

Unfortunately, there remains a drawback to the revised approach. In the case of a stack, the length of the list was precisely equal to the size of the stack (even if the underlying array for the list was slightly larger). With the queue design that we are considering, the situation is worse. We can build a queue that has relatively few elements, yet which are stored in an arbitrarily large list. This occurs, for example, if we repeatedly enqueue a new element and then dequeue another (allowing the front to drift

rightward). Over time, the size of the underlying list would grow to  $O(m)$  where  $m$  is the *total* number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue.

★ 不幸的是，修改后的方法仍然存在缺陷。在栈的情况下，列表的长度正好等于堆栈的大小（即使列表的数组略大）。但是我们考虑的队列设计，情况变糟了。我们可以构建一个具有相对较少元素的队列，而这些队列存储在任意大的列表中。例如，如果我们反复进行新元素入队，然后再让另一个元素出队（允许 `front` 元素向右移动），就会发生这种情况：随着时间的推移，底层列表的大小将增长到  $O(m)$ ，其中  $m$  是自创建队列以来的入队操作的总数，而不是队列中当前的元素数。（即前边都指向 `None`，但是这些指向都是无用的，白白占据了内存空间而永远得不到释放）

This design would have detrimental consequences in applications in which queues have relatively modest size, but which are used for long periods of time. For example, the wait-list for a restaurant might never have more than 30 entries at one time, but over the course of a day (or a week), the overall number of entries would be significantly larger.

★ 这种设计对于队列具有相对适中的尺寸时还算好用，但长期使用的应用将会产生有害的后果。例如，餐厅的等待名单一次可能不会有超过 30 个条目，但是在一天（或一周）的过程中，整个条目数量将显著增加。

#### Using an Array Circularly

★ 使用环数组

In developing a more robust queue implementation, we allow the front of the queue to drift rightward, and we allow the contents of the queue to “wrap around” the end of an underlying array. We assume that our underlying array has fixed length  $N$  that is greater than the actual number of elements in the queue. New elements are enqueued toward the “end” of the current queue, progressing from the front to index  $N-1$  and continuing at index 0, then 1. Figure 6.6 illustrates such a queue with first element  $E$  and last element  $M$ .

★ 在开发更强大的队列时，我们允许队列的前端向右漂移，我们将队列的内容降低为“包围”底层数组的末尾。假设我们的底层数组的固定长度  $N$  大于队列中实际元素数量。新元素入队一直到底层数组的末尾之后，再从前端开始入队，一直到下标  $N-1$ ，如此循环。图 6.6 示出了具有第一个元素  $E$  和最后一个元素  $M$  的队列。（也就是说充分利用起来之前的 `None` 区域，这样会节省内存）

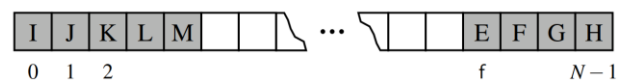


Figure 6.6: Modeling a queue with a circular array that wraps around the end.

Implementing this circular view is not difficult. When we dequeue an element and want to “advance” the front index, we use the arithmetic  $f = (f + 1) \% N$ . Recall that the `%` operator in



Python denotes the **modulo** operator, which is computed by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2, that is,  $\frac{14}{3} = 4\frac{2}{3}$ . So in Python, `14 // 3` evaluates to the quotient 4,

while `14 % 3` evaluates to the remainder 2. The modulo operator is ideal for treating an array circularly. As a concrete example, if we have a list of length 10, and a front index 7, we can advance the front by formally computing  $(7+1) \% 10$ , which is simply 8, as 8 divided by 10 is 0 with a remainder of 8. Similarly, advancing index 8 results in index 9. But when we advance from index 9 (the last one in the array), we compute  $(9+1) \% 10$ , which evaluates to index 0 (as 10 divided by 10 has a remainder of zero).

\* 实施这个循环并不困难。当我们出现一个元素并想要“前进”前导索引时，我们使用算术  $f = (f + 1) \% N$ 。回想一下，Python 中的 % 运算符表示模运算符，它可以返回整数除法的余数。例如，14 除以 3，结果是 4 余 2。所以在 Python 中，`14 % 3` 得 2。模运算符是循环处理 array 的理想选择。作为一个具体的例子，如果我们有一个长度为 10 的列表，目前 front 位于下标 7 的位置，我们可以通过计算  $(7 + 1) \% 10$ （这是简单的 8）来向前推进，因为 `8 // 10 = 8`，而类似地，front 在下标 8 时，计算结果时 9。但是当我们从指数 9（数组中的最后一个）向前推进时，我们计算  $(9 + 1) \% 10$ ，其计算为 0（10 除以 10，重新返回到下标 0）。这样就形成了一个循环。

### A Python Queue Implementation

\* Python 的队列实现

A complete implementation of a queue ADT using a Python list in circular fashion is presented in Code Fragments 6.6 and 6.7. Internally, the queue class maintains the following three instance variables:

\* 代码片段 6.6 和 6.7 中提供了使用循环方式使用 Python 列表的队列 ADT 的完整实现。在内部，队列类支持以下三个实例变量：

- \_data:** is a reference to a list instance with a fixed capacity.  
\* 拥有固定长度的数组。
- \_size:** is an integer representing the current number of elements stored in the queue (as opposed to the length of the data\_list).  
\* 当前实际存储的元素数目。
- \_front:** is an integer that represents the index within \_data of the first element of the queue (assuming the queue is not empty).  
\* front 元素的所在下标。

We initially reserve a list of moderate size for storing data, although the queue formally has size zero. As a technicality, we initialize the front index to zero.

\* 我们最初保留一个中等大小的列表来存储数据，尽管队列的大小为零。我们将前端索引初始化为零。

When front or dequeue are called with no elements in the queue, we raise an instance of the Empty exception, defined in Code Fragment 6.1 for our stack.

\* 当队列中没有元素的时候，如果继续调用 front 与 dequeue 方法，则会抛出 Empty 异常，这在 stack 里面有定义，这里不再重复。

The implementation of `__len__` and `is_empty` are trivial, given knowledge of the size. The implementation of the front method is also simple, as the front index tells us precisely where the desired element is located within the `_data` list, assuming that list is not empty.

\* `__len__` 和 `is_empty` 的实现是微不足道的。front 方法的实现也很简单，因为 front 下标在我们指出了所需元素在 `_data` 列表中的位置。

### Adding and Removing Elements

\* 添加和删除元素

The goal of the enqueue method is to add a new element to the back of the queue. We need to determine the proper index at which to place the new element. Although we do not explicitly maintain an instance variable for the back of the queue, we compute the location of the next opening based on the formula:

\* 入队方法的目标是将一个新的元素添加到队列的后面。我们需要确定放置新元素的下标位置。虽然我们没有显示出队列后面的实例变量，但是我们基于以下公式计算出下一个可以存放数据的位置：

$$\text{avail} = (\text{self._front} + \text{self._size}) \% \text{len}(\text{self._data})$$

Note that we are using the size of the queue as it exists *prior* to the addition of the new element. For example, consider a queue with capacity 10, current size 3, and first element at index 5. The three elements of such a queue are stored at indices 5, 6, and 7. The new element should be placed at index  $(\text{front} + \text{size}) = 8$ . In a case with wrap-around, the use of the modular arithmetic achieves the desired circular semantics. For example, if our hypothetical queue had 3 elements with the first at index 8, our computation of  $(8+3) \% 10$  evaluates to 1, which is perfect since the three existing elements occupy indices 8, 9, and 0.

\* 请注意，在添加新元素之前我们正在使用队列的大小。例如，考虑一个容量为 10，当前大小为 3 的队列，它的 front 在下标位置 5 中。这样一个队列的三个元素存储在 5, 6 和 7 的位置。新元素应该被放置在索引  $(\text{front} + \text{size}) = 8$ 。在这样的情形下，使用模数运算实现了循环需求。例如，如果我们的假设队列有 3 个元素，首先在索引 8，我们的  $(8 + 3) \% 10$  的计算结果为 1，这是完美的，因为三个现有元素占据位置 8, 9 和 0。（精髓所在！）

When the dequeue method is called, the current value of `self._front` designates the index of the value that is to be removed and returned. We keep a local reference to the element that will

be returned, setting `answer = self.data[self._front]` just prior to removing the reference to that object from the list, with the assignment `self._data[self._front] = None`. Our reason for the assignment to `None` relates to Python's mechanism for reclaiming unused space. Internally, Python maintains a count of the number of references that exist to each object. If that count reaches zero, the object is effectively inaccessible, thus the system may reclaim that memory for future use. (For more details, see Section 15.1.2.) Since we are no longer responsible for storing a dequeued element, we remove the reference to it from our list so as to reduce that element's reference count.

\* 当调用 `dequeue` 方法时, `self._front` 所指定的当前数值要被删除和返回。我们保留对要返回的元素的引用, 从列表中删除对该对象的引用之前令 `answer = self._data[self._front]`, 然后再令 `self._data[self._front] = None`。我们分配给 `None` 的原因涉及 Python 的内存回收机制。在内部, Python 保留对每个对象的引用数。如果该计数达到零, 则该对象实际上无法访问, 因此系统可以回收该内存片段以备将来使用。(有关更多详细信息, 请参见第 15.1.2 节) 由于我们不再负责存储出队元素, 因此我们从列表中重新移动引用, 以减少该元素的引用计数。

The second significant responsibility of the `dequeue` method is to update the value of `_front` to reflect the removal of the element, and the presumed promotion of the second element to become the new first. In most cases, we simply want to increment the index by one, but because of the possibility of a wrap-around configuration, we rely on modular arithmetic as originally described on page 242.

\* 出队方法的第二重要任务是更新 `_front` 的价值, 以反映元素的变化, 并将第二个元素推定为新的第一个。在大多数情况下, 我们只需要将索引增加 1, 但是由于可能会进行环绕配置, 所以我们还要依赖于第 242 页所述的模运算。

### Resizing the Queue

\* 队列容量的重整

When `enqueue` is called at a time when the size of the queue equals the size of the underlying list, we rely on a standard technique of doubling the storage capacity of the underlying list. In this way, our approach is similar to the one used when we implemented a `DynamicArray` in Section 5.3.1.

\* 当队列的大小等于基础列表的大小时, 如果再次调用入队方法, 我们就要依赖于将基础列表的存储容量加倍的技术。这样, 我们的方法类似于在 5.3.1 节中实现 `DynamicArray` 时使用的方法。

However, more care is needed in the queue's `resize` utility than was needed in the corresponding method of the `DynamicArray` class. After creating a temporary reference to the old list of values, we allocate a new list that is twice the size and copy references from the old list to the new list. While transferring the contents, we intentionally realign the front of the queue with index 0 in the new array, as shown in Figure 6.7. This realignment is not purely cosmetic. Since the modular arithmetic depends on

the size of the array, our state would be flawed had we transferred each element to its same index in the new array.

\* 但是, 在 `DynamicArray` 类的相应方法中, 需要更加注意队列的调整大小。在创建对旧列表值的临时引用之后, 我们分配一个新列表, 它是原来列表的两倍大小, 并将引用从旧列表复制到新列表。在传输内容的同时, 我们有意在新数组中使用下标 0 重新排列队列的 `front`, 如图 6.7 所示。这种调整不是纯粹的闹着玩。由于模数运算依赖于数组的大小, 如果我们将每个元素转移到新数组中的相同索引, 那么我们的状态将会被忽略 (从而运算量将会相对减少)。

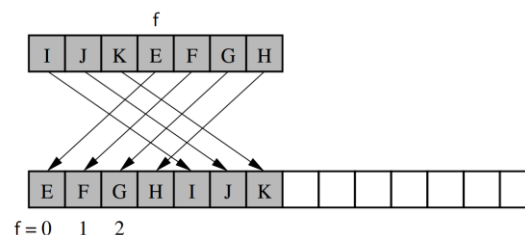


Figure 6.7: Resizing the queue, while realigning the front element with index 0.

### Shrinking the Underlying Array

\* 减少底层数组

A desirable property of a queue implementation is to have its space usage be  $\Theta(n)$  where  $n$  is the current number of elements in the queue. Our `ArrayQueue` implementation, as given in Code Fragments 6.6 and 6.7, does not have this property. It expands the underlying array when `enqueue` is called with the queue at full capacity, but the `dequeue` implementation never shrinks the underlying array. As a consequence, the capacity of the underlying array is proportional to the maximum number of elements that have ever been stored in the queue, not the current number of elements.

\* 队列实现的期望属性是使其空间使用为  $\Theta(n)$ , 其中  $n$  是队列中当前的元素数。我们在代码片段 6.6 和 6.7 中给出的 `ArrayQueue` 实现没有此属性。当队列以满容量调用队列时, 它扩展底层数组, 但是出队实现却不缩小底层数组。作为一个规则, 底层数组的容量与存储在队列中的元素的最大数目成比例, 而不是元素的当前数量。

We discussed this very issue on page 200, in the context of dynamic arrays, and in subsequent Exercises C-5.16 through C-5.20 of that chapter. A robust approach is to reduce the array to half of its current size, whenever the number of elements stored in it falls below *one fourth* of its capacity. We can implement this strategy by adding the following two lines of code in our `dequeue` method, just after reducing `self.size` at line 38 of Code Fragment 6.6, to reflect the loss of an element.

\* 我们在第 200 页上讨论了这个非常棘手的问题, 动态数组的结构以及本章后续练习 C-5.16 至 C-5.20 都有所涉及。一个稳健的方法是, 每当存储在其中的元素数量低于其容量的四分之一, 将阵列减少到其当前大小的一半。我们可以通过在我们的出队方法中添加以下两行代码, 在减少自

身之后实现这一策略。代码片段 6.6 的第 38 行的大小，以反映元素的减少。

```
if 0 < self._size < len(self.data) // 4:
    self.resize(len(self.data) // 2)
```

Analyzing the Array-Based Queue Implementation

\* 对基于数组的队列的分析

Table 6.3 describes the performance of our array-based implementation of the queue ADT, assuming the improvement described above for occasionally shrinking the size of the array. With the exception of the `resize` utility, all of the methods rely on a constant number of statements involving arithmetic operations, comparisons, and assignments. Therefore, each method runs in worst-case  $O(1)$  time, except for `enqueue` and `dequeue`, which have *amortized* bounds of  $O(1)$  time, for reasons similar to those given in Section 5.3.

\* 表 6.3 描述了我们基于数组的队列 ADT 实现的性能，假设上述针对偶尔缩小阵列大小的改进。除了 `resize` 实用程序之外，所有方法都依赖于常量数量的语句，包括算术运算，比较和赋值。因此，由于类似于 5.3 节中给出的原因，每种方法都运行在最坏情况  $O(1)$  时间，排除了入队和出队，其具有  $O(1)$  时间复杂度。

Operation	Running Time
<code>Q.enqueue(e)</code>	$O(1)^*$
<code>Q.dequeue()</code>	$O(1)^*$
<code>Q.first()</code>	$O(1)$
<code>Q.is_empty()</code>	$O(1)$
<code>len(Q)</code>	$O(1)$

\*amortized

**Table 6.3:** Performance of an array-based implementation of a queue. The bounds for `enqueue` and `dequeue` are amortized due to the resizing of the array. The space usage is  $O(n)$ , where  $n$  is the current number of elements in the queue.



### 6.3 Double-Ended Queues

#### \* 双端队列

We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a *double-ended queue*, or deque, which is usually pronounced “deck” to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation “D.Q.”

\* 接下来我们考虑一个队列状数据结构，它支持在队列的前面和后面插入和删除。这种结构被称为双端队列，或者 deque，通常发音为 “deck”，以避免与正常队列 ADT 的出队方法 dequeue（发音 “D.Q.”）矛盾。

The deque abstract data type is more general than both the stack and the queue ADTs. The extra generality can be useful in some applications. For example, we described a restaurant using a queue to maintain a waitlist. Occasionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will re-insert the person at the first position in the queue. It may also be that a customer at the end of the queue may grow impatient and leave the restaurant. (We will need an even more general data structure if we want to model customers leaving the queue from other positions.)

\* deque 抽象数据类型比堆栈和队列 ADT 更通用。额外的通用性在一些应用中可能是有用的。例如，我们描述了使用队列来记录餐厅的排队情况情形。有时发现一张桌子不可用，第一个人可能会从队列中移除；通常情况下，餐厅将把这个顾客重新插入到队列中的第一个位置的。可能的是，队列结束时的客户可能会不耐烦地离开餐厅。（如果我们想模拟从其他职位离开队列的客户，我们将需要一个更一般的数据结构。）

#### 6.3.1 The Deque Abstract Data Type

##### \* 6.3.1 节 双端队列的抽象数据结构

To provide a symmetrical abstraction, the deque ADT is defined so that deque D supports the following methods:

\* 为了提供对称抽象，我们定义了 deque 的 ADT，一个 deque D 支持以下方法：

**D.add\_first(e):** Add element e to the front of deque D.

**D.add\_last(e):** Add element e to the back of deque D.

**D.delete\_first():** Remove and return the first element from deque D; an error occurs if the deque is empty.

**D.delete\_last():** Remove and return the last element from deque D; an error occurs if the deque is empty.

Additionally, the deque ADT will include the following accessors:

\* 此外，deque ADT 将包括以下访问器：

**D.first():** Return (but do not remove) the first element of deque D; an error occurs if the deque is empty.

\* 返回（但是并不删除）D 的第一个元素；如果 D 为空，就会抛出异常。

**D.last():** Return (but do not remove) the last element of deque D; an error occurs if the deque is empty.

\* 返回（但是不删除）D 的最后一个元素；如果 D 为空，就会抛出异常。

**D.is\_empty():** Return True if deque D does not contain any elements.

\* 如果 D 不包含任何元素，那么就返回 True。

**len(D):** Return the number of elements in deque D; in Python, we implement this with the special method `__len__`.

\* 返回 D 中的元素数目。在 Python 中，我们通过特殊的 `__len__` 方法来实现这一目的。

**Example 6.5:** The following table shows a series of operations and their effects on an initially empty deque D of integers.

Operation	Return Value	Deque
D.add_last(5)	—	[5]
D.add_first(3)	—	[3, 5]
D.add_first(7)	—	[7, 3, 5]
D.first()	7	[7, 3, 5]
D.delete_last()	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last()	3	[7]
D.delete_last()	7	[]
D.add_first(6)	—	[6]
D.last()	6	[6]
D.add_first(8)	—	[8, 6]
D.is_empty()	False	[8, 6]
D.last()	6	[8, 6]

### 6.3.2 Implementing a Deque with a Circular Array

We can implement the deque ADT in much the same way as the ArrayQueue class provided in Code Fragments 6.6 and 6.7 of Section 6.2.2 (so much so that we leave the details of an ArrayDeque implementation to Exercise P-6.32). We recommend maintaining the same three instance variables: `_data`, `_size`, and `_front`. Whenever we need to know the index of the back of the deque, or the first available slot beyond the back of the deque, we use modular arithmetic for the computation. For example, our implementation of the `last()` method uses the index

\* 我们可以以与 6.2.2 节的 Code Fragments 6.6 和 6.7 中提供的 ArrayQueue 类相同的方式实现 deque ADT（这样我们将一个 ArrayDeque 实现的尾部删除留到练习 P-6.32）。我们建议维护保留的三个变量：`_data`、`_size` 和 `_front`。每当我们知道 deque 的下标时，或者超出 deque 背面的第一个可用位置时，我们使用模数运算来进行计算。例如，我们实现的 `last()` 方法，就是使用了索引：

```
back = (self._front + self._size - 1) % len(self._data)
```

Our implementation of the `ArrayDeque.add_last` method is essentially the same as that for `ArrayQueue.enqueue`, including the reliance on a `resize` utility. Likewise, the implementation of

the `ArrayDeque.delete_first` method is the same as `ArrayQueue.dequeue`. Implementations of `add_first` and `delete_last` use similar techniques. One subtlety is that a call to `add_first` may need to wrap around the beginning of the array, so we rely on modular arithmetic to circularly decrement the index, as

\* 我们对 `ArrayDeque.add_last` 方法的实现与 `ArrayQueue.enqueue` 的实现基本相同，包括依赖于 `resize` 实用程序。同样，`ArrayDeque.delete_first` 方法的实现与 `ArrayQueue.dequeue` 相同。`add_first` 和 `delete_last` 的实现使用类似的方法技术。一个微妙之处在于，`add_first` 的调用可能需要围绕数组的开头，所以我们依靠模运算来循环递减索引，如

```
self._front = (self._front - 1) % len(self._data)
# cyclic shift
```

The efficiency of an `ArrayDeque` is similar to that of an `ArrayQueue`, with all operations having  $O(1)$  running time, but with that bound being amortized for operations that may change the size of the underlying list.

\* `ArrayDeque` 的效率与 `ArrayQueue` 的效率类似，所有操作都具有  $O(1)$  运行时间，但是对于可能更改底层列表大小的操作，该操作将被分摊。

6.3.3 Deques in the Python Collections Module

\*

An implementation of a deque class is available in Python’s standard collections module. A summary of the most commonly used behaviors of the collections.deque class is given in Table 6.4. It uses more asymmetric nomenclature than our ADT.

\* Python 标准集合模块中提供了 deque 类的实现。表 6.4 中给出了 collections.deque 类最常用行为的总结。它比我们的 ADT 使用更不对称的非标记。

Our Deque ADT	collections.deque	Description
len(D)	len(D)	number of elements
D.add_first()	D.appendleft()	add to beginning
D.add_last()	D.append()	add to end
D.delete_first()	D.popleft()	remove from beginning
D.delete_last()	D.pop()	remove from end
D.first()	D[0]	access first element
D.last()	D[-1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear()	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Table 6.4: Comparison of our deque ADT and the collections.deque class.

The collections.deque interface was chosen to be consistent with established naming conventions of Python’s list class, for which append and pop are presumed to act at the end of the list. Therefore, appendleft and popleft designate an operation at the beginning of the list. The library deque also mimics a list in that it is an indexed sequence, allowing arbitrary access or modification using the D[j] syntax.

\* 假定其 append 和 pop 行为在列表的末尾,那么 collections.deque 接口与 Python 的列表类的已建立的命名约

定一致。因此, appendleft 和 popleft 在列表开头设计一个操作。库 deque 还模拟了一个列表,因为它是一个索引序列,允许使用 D[j]语法进行任意访问或修改。

The library deque constructor also supports an optional maxlen parameter to force a fixed-length deque. However, if a call to append at either end is invoked when the deque is full, it does not throw an error; instead, it causes one element to be dropped from the opposite side. That is, calling appendleft when the deque is full causes an implicit pop from the right side to make room for the new element.

\* 库 deque 构造函数还支持一个可选的 maxlen 参数来强制固定长度的 deque。但是,如果在 deque 满的时候调用在任一端附加的调用,则不会引发错误;相反,它导致一个元素从相对侧被挤下来。也就是说,当 deque 完整时,调用 appendleft 会导致右侧弹出元素以便腾出空间。

The current Python distribution implements collections.deque with a hybrid approach that uses aspects of circular arrays, but organized into blocks that are themselves organized in a doubly linked list (a data structure that we will introduce in the next chapter). The deque class is formally documented to guarantee  $O(1)$ -time operations at either end, but  $O(n)$ -time worst-case operations when using index notation near the middle of the deque.

\* 目前的 Python 发行版使用混合方法实现了 collections.deque,该方法使用循环数组的方面,但是被组织成一个自己组织在一个双向链表中的块(我们将在下一章中介绍的数据结构)。deque 类被正式记录,以保证在任何一端的  $O(1)$ 时间操作,但是在使用靠近 deque 中间的索引符号时,有  $O(n)$ 的最坏时间复杂度。

END

## 六、实验体会

有关代码的分析与体会请看实验记录与实验结果分析中的代码分析段落。

总结来看，有以下几点比较有新意：

1. 适配器模式；可以通过这个模式将一个类伪装成另一个类，但是外界看不出任何变化；
2. 栈与队列这两种逻辑结构的天然应用；
3. 模运算的周期性。

具体的分析请结合导航面板的标题栏进行查阅。

## 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