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

*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

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 $0 \le k \le n$ 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

```
1  def insert(self, k, value):
2    """Insert value at index k, shifting subsequent values rightward."""
3    # (for simplicity, we assume 0 <= k <= n in this verion)
4    if self._n == self._capacity:                # not enough room
5      self._resize(2 * self._capacity)           # so double capacity
6    for j in range(self._n, k, -1):              # shift rightmost first
7      self._A[j] = self._A[j-1]
8    self._A[k] = value                           # store newest element
9    self._n += 1
```

Code Fragment 5.5: Implementation of insert for our DynamicArray class.

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.

5.4.1 Python's List and Tuple Classes

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.

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

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

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.

5.6 Multidimensional Data Sets

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.

	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

$$\begin{array}{ccccc} 22 & 18 & 709 & 5 & 33 \\ 45 & 32 & 830 & 120 & 750 \\ 4 & 880 & 45 & 66 & 61 \end{array}$$

might be stored in Python as follows.

```
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.

```
1  class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with a binary heap."""
3    #------------------------------ nonpublic behaviors ------------------------------
4    def _parent(self, j):
5      return (j-1) // 2
6
7    def _left(self, j):
8      return 2*j + 1
9
10   def _right(self, j):
11     return 2*j + 2
12
13   def _has_left(self, j):
14     return self._left(j) < len(self._data)      # index beyond end of list?
15
16   def _has_right(self, j):
17     return self._right(j) < len(self._data)     # index beyond end of list?
18
19   def _swap(self, i, j):
20     """Swap the elements at indices i and j of array."""
21     self._data[i], self._data[j] = self._data[j], self._data[i]
22
23   def _upheap(self, j):
24     parent = self._parent(j)
25     if j > 0 and self._data[j] < self._data[parent]:
26       self._swap(j, parent)
27       self._upheap(parent)                      # recur at position of parent
28
29   def _downheap(self, j):
30     if self._has_left(j):
31       left = self._left(j)
32       small_child = left                        # although right may be smaller
33       if self._has_right(j):
34         right = self._right(j)
35         if self._data[right] < self._data[left]:
36           small_child = right
37       if self._data[small_child] < self._data[j]:
38         self._swap(j, small_child)
39         self._downheap(small_child)             # recur at position of small child
```

Code Fragment 9.4: An implementation of a priority queue using an array-based heap (continued in Code Fragment 9.5). The extends the PriorityQueueBase class from Code Fragment 9.1.

These assignment statements establish identifier data as an alias for grades and target as a name for the string literal 'A'. (See Figure 1.7.)

Figure 1.7: A portrayal of parameter passing in Python, for the function call count(grades, 'A'). Identifiers data and target are formal parameters defined within the local scope of the count function.

The communication of a return value from the function back to the caller is similarly implemented as an assignment. Therefore, with our sample invocation of prizes = count(grades, 'A'), the identifier prizes in the caller's scope is assigned to the object that is identified as n in the return statement within our function body.

An advantage to Python's mechanism for passing information to and from a function is that objects are not copied. This ensures that the invocation of a function is efficient, even in a case where a parameter or return value is a complex object.

Mutable Parameters

Python's parameter passing model has additional implications when a parameter is a mutable object. Because the formal parameter is an alias for the actual parameter, the body of the function may interact with the object in ways that change its state. Considering again our sample invocation of the count function, if the body of the function executes the command data.append('F'), the new entry is added to the end of the list identified as data within the function, which is one and the same as the list known to the caller as grades. As an aside, we note that reassigning a new value to a formal parameter with a function body, such as by setting data = [ ], does not alter the actual parameter; such a reassignment simply breaks the alias.

Our hypothetical example of a count method that appends a new element to a list lacks common sense. There is no reason to expect such a behavior, and it would be quite a poor design to have such an unexpected effect on the parameter. There are, however, many legitimate cases in which a function may be designed (and clearly documented) to modify the state of a parameter. As a concrete example, we present the following implementation of a method named scale that's primary purpose is to multiply all entries of a numeric data set by a given factor.

```
def scale(data, factor):
  for j in range(len(data)):
    data[j] *= factor
```

Decorate-Sort-Undecorate Design Pattern

Python's support for a key function when sorting is implemented using what is known as the ***decorate-sort-undecorate design pattern***. It proceeds in 3 steps:

1. Each element of the list is temporarily replaced with a "decorated" version that includes the result of the key function applied to the element.
2. The list is sorted based upon the natural order of the keys (Figure 12.16).
3. The decorated elements are replaced by the original elements.

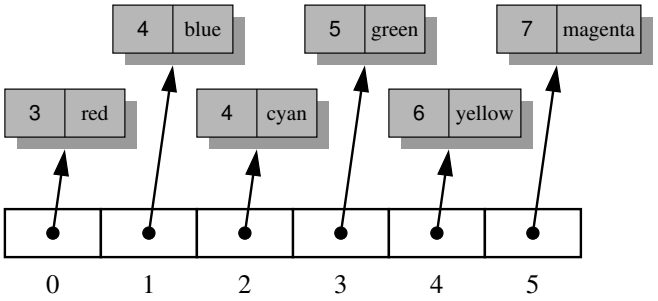

Figure 12.16: A list of "decorated" strings, using their lengths as decoration. This list has been sorted by those keys.

Although there is already built-in support for Python, if we were to implement such a strategy ourselves, a natural way to represent a "decorated" element is using the same composition strategy that we used for representing key-value pairs within a priority queue. Code Fragment 9.1 of Section 9.2.1 includes just such an _Item class, defined so that the $<$ operator for items relies upon the given keys. With such a composition, we could trivially adapt any sorting algorithm to use the decorate-sort-undecorate pattern, as demonstrated in Code Fragment 12.8 with merge-sort.

```
1  def decorated_merge_sort(data, key=None):
2    """Demonstration of the decorate-sort-undecorate pattern."""
3    if key is not None:
4      for j in range(len(data)):
5        data[j] = _Item(key(data[j]), data[j])      # decorate each element
6    merge_sort(data)                                # sort with existing algorithm
7    if key is not None:
8      for j in range(len(data)):
9        data[j] = data[j]._value                    # undecorate each element
```

Code Fragment 12.8: An approach for implementing the decorate-sort-undecorate pattern based upon the array-based merge-sort of Code Fragment 12.1. The _Item class is identical to that which was used in the PriorityQueueBase class. (See Code Fragment 9.1.)

Documentation

Python provides integrated support for embedding formal documentation directly in source code using a mechanism known as a ***docstring***. Formally, any string literal that appears as the *first* statement within the body of a module, class, or function (including a member function of a class) will be considered to be a docstring. By convention, those string literals should be delimited within triple quotes ("""). As an example, our version of the scale function from page 25 could be documented as follows:

```
def scale(data, factor):
  """Multiply all entries of numeric data list by the given factor."""
  for j in range(len(data)):
    data[j] *= factor
```

It is common to use the triple-quoted string delimiter for a docstring, even when the string fits on a single line, as in the above example. More detailed docstrings should begin with a single line that summarizes the purpose, followed by a blank line, and then further details. For example, we might more clearly document the scale function as follows:

```
def scale(data, factor):
  """Multiply all entries of numeric data list by the given factor.

  data    an instance of any mutable sequence type (such as a list)
          containing numeric elements

  factor  a number that serves as the multiplicative factor for scaling
  """
  for j in range(len(data)):
    data[j] *= factor
```

A docstring is stored as a field of the module, function, or class in which it is declared. It serves as documentation and can be retrieved in a variety of ways. For example, the command help(x), within the Python interpreter, produces the documentation associated with the identified object x. An external tool named pydoc is distributed with Python and can be used to generate formal documentation as text or as a Web page. Guidelines for *authoring* useful docstrings are available at:

`http://www.python.org/dev/peps/pep-0257/`

In this book, we will try to present docstrings when space allows. Omitted docstrings can be found in the online version of our source code.

```
40   def enqueue(self, e):
41     """Add an element to the back of queue."""
42     if self._size == len(self._data):
43       self._resize(2 * len(self.data))          # double the array size
44     avail = (self._front + self._size) % len(self._data)
45     self._data[avail] = e
46     self._size += 1
47
48   def _resize(self, cap):                       # we assume cap >= len(self)
49     """Resize to a new list of capacity >= len(self)."""
50     old = self._data                            # keep track of existing list
51     self._data = [None] * cap                   # allocate list with new capacity
52     walk = self._front
53     for k in range(self._size):                 # only consider existing elements
54       self._data[k] = old[walk]                 # intentionally shift indices
55       walk = (1 + walk) % len(old)              # use old size as modulus
56     self._front = 0                             # front has been realigned
```

Code Fragment 6.7: Array-based implementation of a queue (continued from Code Fragment 6.6).

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.

Adding and Removing Elements

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:

```
avail = (self._front + self._size) % len(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.

Using the Composition Pattern

We wish to implement a favorites list by making use of a PositionalList for storage. If elements of the positional list were simply elements of the favorites list, we would be challenged to maintain access counts and to keep the proper count with the associated element as the contents of the list are reordered. We use a general object-oriented design pattern, the ***composition pattern***, in which we define a single object that is composed of two or more other objects. Specifically, we define a nonpublic nested class, _Item, that stores the element and its access count as a single instance. We then maintain our favorites list as a PositionalList of *item* instances, so that the access count for a user's element is embedded alongside it in our representation. (An _Item is never exposed to a user of a FavoritesList.)

```
1  class FavoritesList:
2    """List of elements ordered from most frequently accessed to least."""
3
4    #------------------------------ nested _Item class ------------------------------
5    class _Item:
6      __slots__ = '_value', '_count'              # streamline memory usage
7      def __init__(self, e):
8        self._value = e                           # the user's element
9        self._count = 0                           # access count initially zero
10
11   #------------------------------ nonpublic utilities ------------------------------
12   def _find_position(self, e):
13     """Search for element e and return its Position (or None if not found)."""
14     walk = self._data.first( )
15     while walk is not None and walk.element( )._value != e:
16       walk = self._data.after(walk)
17     return walk
18
19   def _move_up(self, p):
20     """Move item at Position p earlier in the list based on access count."""
21     if p != self._data.first( ):                  # consider moving...
22       cnt = p.element( )._count
23       walk = self._data.before(p)
24       if cnt > walk.element( )._count:           # must shift forward
25         while (walk != self._data.first( ) and
26                cnt > self._data.before(walk).element( )._count):
27           walk = self._data.before(walk)
28         self._data.add_before(walk, self._data.delete(p))     # delete/reinsert
```

Code Fragment 7.18: Class FavoritesList. (Continues in Code Fragment 7.19.)

```
1  class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with a sorted list."""
3
4    def __init__(self):
5      """Create a new empty Priority Queue."""
6      self._data = PositionalList( )
7
8    def __len__(self):
9      """Return the number of items in the priority queue."""
10     return len(self._data)
11
12   def add(self, key, value):
13     """Add a key-value pair."""
14     newest = self._Item(key, value)              # make new item instance
15     walk = self._data.last( )          # walk backward looking for smaller key
16     while walk is not None and newest < walk.element( ):
17       walk = self._data.before(walk)
18     if walk is None:
19       self._data.add_first(newest)               # new key is smallest
20     else:
21       self._data.add_after(walk, newest)         # newest goes after walk
22
23   def min(self):
24     """Return but do not remove (k,v) tuple with minimum key."""
25     if self.is_empty( ):
26       raise Empty('Priority queue is empty.')
27     p = self._data.first( )
28     item = p.element( )
29     return (item._key, item._value)
30
31   def remove_min(self):
32     """Remove and return (k,v) tuple with minimum key."""
33     if self.is_empty( ):
34       raise Empty('Priority queue is empty.')
35     item = self._data.delete(self._data.first( ))
36     return (item._key, item._value)
```

Code Fragment 9.3: An implementation of a priority queue using a sorted list. The parent class PriorityQueueBase is given in Code Fragment 9.1, and the PositionalList class is from Section 7.4.