```
1  >>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
2  >>> print(numbers) Enter
3  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4  >>> print(numbers[1:8:2]) Enter
5  [2, 4, 6, 8]
6  >>>
```

In the slicing expression in line 4, the third number inside the brackets is the step value. A step value of 2, as used in this example, causes the slice to contain every second element from the specified range in the list.

You can also use negative numbers as indexes in slicing expressions to reference positions relative to the end of the list. Python adds a negative index to the length of a list to get the position referenced by that index. The following interactive mode session shows an example:

```
1  >>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] Enter
2  >>> print(numbers) Enter
3  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4  >>> print(numbers[-5:]) Enter
5  [6, 7, 8, 9, 10]
6  >>>
```

NOTE: Invalid indexes do not cause slicing expressions to raise an exception. For example:

- If the *end* index specifies a position beyond the end of the list, Python will use the length of the list instead.
- If the *start* index specifies a position before the beginning of the list, Python will use 0 instead.
- If the *start* index is greater than the *end* index, the slicing expression will return an empty list.

Checkpoint

7.9 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[1:3]
print(my_list)
```

7.10 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[1:]
print(my_list)
```

7.11 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[:1]
print(my_list)
```

7.12 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[:]
print(my_list)
```

7.13 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
my_list = numbers[-3:]
print(my_list)
```

7.4 Finding Items in Lists with the `in` Operator

CONCEPT: You can search for an item in a list using the `in` operator.

In Python you can use the `in` operator to determine whether an item is contained in a list. Here is the general format of an expression written with the in operator to search for an item in a list:

```
item in list
```

In the general format, *`item`* is the item for which you are searching, and *`list`* is a list. The expression returns true if *`item`* is found in the *`list`* or false otherwise. Program 7-2 shows an example.

Program 7-2 (in_list.py)

```
1  # This program demonstrates the in operator
2  # used with a list.
3
4  def main():
5      # Create a list of product numbers.
6      prod_nums = ['V475', 'F987', 'Q143', 'R688']
7
8      # Get a product number to search for.
9      search = input('Enter a product number: ')
10
11     # Determine whether the product number is in the list.
12     if search in prod_nums:
13         print(search, 'was found in the list.')
14     else:
15         print(search, 'was not found in the list.')
16
17 # Call the main function.
18 main()
```

Program Output (with input shown in bold)

```
Enter a product number: Q143 Enter
Q143 was found in the list.
```

Program Output (with input shown in bold)

```
Enter a product number: B000 Enter
B000 was not found in the list.
```

The program gets a product number from the user in line 9 and assigns it to the `search` variable. The `if` statement in line 12 determines whether `search` is in the `prod_nums` list.

You can use the `not in` operator to determine whether an item is *not* in a list. Here is an example:

```
if search not in prod_nums:
    print(search, 'was not found in the list.')
else:
    print(search, 'was found in the list.')
```

Checkpoint

7.14 What will the following code display?

```
names = ['Jim', 'Jill', 'John', 'Jasmine']
if 'Jasmine' not in names:
    print('Cannot find Jasmine.')
else:
    print("Jasmine's family:")
    print(names)
```

7.5 List Methods and Useful Built-in Functions

CONCEPT: **Lists have numerous methods that allow you to work with the elements that they contain. Python also provides some built-in functions that are useful for working with lists.**

Lists have numerous methods that allow you to add elements, remove elements, change the ordering of elements, and so forth. We will look at a few of these methods,[1] which are listed in Table 7-1.

The `append` Method

The `append` method is commonly used to add items to a list. The item that is passed as an argument is appended to the end of the list's existing elements. Program 7-3 shows an example.

[1] We do not cover all of the list methods in this book. For a description of all of the list methods, see the Python documentation at www.python.org.

Table 7-1 A few of the list methods

Method	Description
`append(item)`	Adds *item* to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts *item* into the list at the specified *index*. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
`sort()`	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
`remove(item)`	Removes the first occurrence of *item* from the list. A `ValueError` exception is raised if item is not found in the list.
`reverse()`	Reverses the order of the items in the list.

Program 7-3 (list_append.py)

```
1  # This program demonstrates how the append
2  # method can be used to add items to a list.
3
4  def main():
5      # First, create an empty list.
6      name_list = []
7
8      # Create a variable to control the loop.
9      again = 'y'
10
11     # Add some names to the list.
12     while again == 'y':
13         # Get a name from the user.
14         name = input('Enter a name: ')
15
16         # Append the name to the list.
17         name_list.append(name)
18
19         # Add another one?
20         print('Do you want to add another name?')
21         again = input('y = yes, anything else = no: ')
22         print()
23
```

```
24      # Display the names that were entered.
25      print('Here are the names you entered.')
26
27      for name in name_list:
28          print(name)
29
30  # Call the main function.
31  main()
```

Program Output (with input shown in bold)

```
Enter a name: Kathryn [Enter]
Do you want to add another name?
y = yes, anything else = no: y [Enter]

Enter a name: Chris [Enter]
Do you want to add another name?
y = yes, anything else = no: y [Enter]

Enter a name: Kenny [Enter]
Do you want to add another name?
y = yes, anything else = no: y [Enter]

Enter a name: Renee [Enter]
Do you want to add another name?
y = yes, anything else = no: n [Enter]

Here are the names you entered.
Kathryn
Chris
Kenny
Renee
```

Notice the statement in line 6:

```
name_list = []
```

This statement creates an empty list (a list with no elements) and assigns it to the `name_list` variable. Inside the loop, the `append` method is called to build the list. The first time the method is called, the argument passed to it will become element 0. The second time the method is called, the argument passed to it will become element 1. This continues until the user exits the loop.

The `index` Method

Earlier you saw how the `in` operator can be used to determine whether an item is in a list. Sometimes you need to know not only whether an item is in a list, but where it is located. The `index` method is useful in these cases. You pass an argument to the `index` method, and it returns the index of the first element in the list containing that item. If the item is not found in the list, the method raises a `ValueError` exception. Program 7-4 demonstrates the `index` method.

Program 7-4 (index_list.py)

```
1  # This program demonstrates how to get the
2  # index of an item in a list and then replace
3  # that item with a new item.
4
5  def main():
6      # Create a list with some items.
7      food = ['Pizza', 'Burgers', 'Chips']
8
9      # Display the list.
10     print('Here are the items in the food list:')
11     print(food)
12
13     # Get the item to change.
14     item = input('Which item should I change? ')
15
16     try:
17         # Get the item's index in the list.
18         item_index = food.index(item)
19
20         # Get the value to replace it with.
21         new_item = input('Enter the new value: ')
22
23         # Replace the old item with the new item.
24         food[item_index] = new_item
25
26         # Display the list.
27         print('Here is the revised list:')
28         print(food)
29     except ValueError:
30         print('That item was not found in the list.')
31
32 # Call the main function.
33 main()
```

Program Output (with input shown in bold)

```
Here are the items in the food list:
['Pizza', 'Burgers', 'Chips']
Which item should I change? Burgers Enter
Enter the new value: Pickles Enter
Here is the revised list:
['Pizza', 'Pickles', 'Chips']
```

The elements of the `food` list are displayed in line 11, and in line 14 the user is asked which item he or she wants to change. Line 18 calls the `index` method to get the index of the item.

Line 21 gets the new value from the user, and line 24 assigns the new value to the element holding the old value.

The `insert` Method

The `insert` method allows you to insert an item into a list at a specific position. You pass two arguments to the `insert` method: an index specifying where the item should be inserted and the item that you want to insert. Program 7-5 shows an example.

Program 7-5 (insert_list.py)

```
1  # This program demonstrates the insert method.
2
3  def main():
4      # Create a list with some names.
5      names = ['James', 'Kathryn', 'Bill']
6
7      # Display the list.
8      print('The list before the insert:')
9      print(names)
10
11     # Insert a new name at element 0.
12     names.insert(0, 'Joe')
13
14     # Display the list again.
15     print('The list after the insert:')
16     print(names)
17
18 # Call the main function.
19 main()
```

Program Output

```
The list before the insert:
['James', 'Kathryn', 'Bill']
The list after the insert:
['Joe', 'James', 'Kathryn', 'Bill']
```

The `sort` Method

The `sort` method rearranges the elements of a list so they appear in ascending order (from the lowest value to the highest value). Here is an example:

```
my_list = [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
print('Original order:', my_list)
my_list.sort()
print('Sorted order:', my_list)
```

When this code runs it will display the following:

```
Original order: [9, 1, 0, 2, 8, 6, 7, 4, 5, 3]
Sorted order: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Here is another example:

```
my_list = ['beta', 'alpha', 'delta', 'gamma']
print('Original order:', my_list)
my_list.sort()
print('Sorted order:', my_list)
```

When this code runs it will display the following:

```
Original order: ['beta', 'alpha', 'delta', 'gamma']
Sorted order: ['alpha', 'beta', 'delta', 'gamma']
```

The `remove` Method

The `remove` method removes an item from the list. You pass an item to the method as an argument, and the first element containing that item is removed. This reduces the size of the list by one element. All of the elements after the removed element are shifted one position toward the beginning of the list. A `ValueError` exception is raised if the item is not found in the list. Program 7-6 demonstrates the method.

Program 7-6 (remove_item.py)

```
1  # This program demonstrates how to use the remove
2  # method to remove an item from a list.
3
4  def main():
5      # Create a list with some items.
6      food = ['Pizza', 'Burgers', 'Chips']
7
8      # Display the list.
9      print('Here are the items in the food list:')
10     print(food)
11
12     # Get the item to change.
13     item = input('Which item should I remove? ')
14
15     try:
16         # Remove the item.
17         food.remove(item)
18
19         # Display the list.
20         print('Here is the revised list:')
21         print(food)
22
23     except ValueError:
```

```
24          print('That item was not found in the list.')
25
26  # Call the main function.
27  main()
```

Program Output (with input shown in bold)

```
Here are the items in the food list:
['Pizza', 'Burgers', 'Chips']
Which item should I remove? Burgers Enter
Here is the revised list:
['Pizza', 'Chips']
```

The `reverse` Method

The `reverse` method simply reverses the order of the items in the list. Here is an example:

```
my_list = [1, 2, 3, 4, 5]
print('Original order:', my_list)
my_list.reverse()
print('Reversed:', my_list)
```

This code will display the following:

```
Original order: [1, 2, 3, 4, 5]
Reversed: [5, 4, 3, 2, 1]
```

The `del` Statement

The `remove` method that you saw earlier removes a specific item from a list, if that item is in the list. Some situations might require that you remove an element from a specific index, regardless of the item that is stored at that index. This can be accomplished with the `del` statement. Here is an example of how to use the `del` statement:

```
my_list = [1, 2, 3, 4, 5]
print('Before deletion:', my_list)
del my_list[2]
print('After deletion:', my_list)
```

This code will display the following:

```
Before deletion: [1, 2, 3, 4, 5]
After deletion: [1, 2, 4, 5]
```

The `min` and `max` Functions

Python has two built-in functions named `min` and `max` that work with sequences. The `min` function accepts a sequence, such as a list, as an argument and returns the item that has the lowest value in the sequence. Here is an example:

```
my_list = [5, 4, 3, 2, 50, 40, 30]
print('The lowest value is', min(my_list))
```

This code will display the following:

```
The lowest value is 2
```

The `max` function accepts a sequence, such as a list, as an argument and returns the item that has the highest value in the sequence. Here is an example:

```
my_list = [5, 4, 3, 2, 50, 40, 30]
print('The highest value is', max(my_list))
```

This code will display the following:

```
The highest value is 50
```

Checkpoint

7.15 What is the difference between calling a list's `remove` method and using the `del` statement to remove an element?

7.16 How do you find the lowest and highest values in a list?

7.17 Assume the following statement appears in a program:

```
names = []
```

Which of the following statements would you use to add the string 'Wendy' to the list at index 0? Why would you select this statement instead of the other?

```
a. names[0] = 'Wendy'
b. names.append('Wendy')
```

7.18 Describe the following list methods:

```
a. index
b. insert
c. sort
d. reverse
```

7.6 Copying Lists

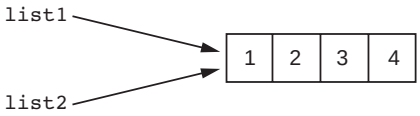
CONCEPT: To make a copy of a list, you must copy the list's elements.

Recall that in Python, assigning one variable to another variable simply makes both variables reference the same object in memory. For example, look at the following code:

```
# Create a list.
list1 = [1, 2, 3, 4]
# Assign the list to the list2 variable.
list2 = list1
```

After this code executes, both variables `list1` and `list2` will reference the same list in memory. This is shown in Figure 7-4.

Figure 7-4 `list1` and `list2` reference the same list



To demonstrate this, look at the following interactive session:

```
1   >>> list1 = [1, 2, 3, 4] Enter
2   >>> list2 = list1 Enter
3   >>> print(list1) Enter
4   [1, 2, 3, 4]
5   >>> print(list2) Enter
6   [1, 2, 3, 4]
7   >>> list1[0] = 99 Enter
8   >>> print(list1) Enter
9   [99, 2, 3, 4]
10  >>> print(list2) Enter
11  [99, 2, 3, 4]
12  >>>
```

Let's take a closer look at each line:

- In line 1 we create a list of integers and assign the list to the `list1` variable.
- In line 2 we assign `list1` to `list2`. After this, both `list1` and `list2` reference the same list in memory.
- In line 3 we print the list referenced by `list1`. The output of the `print` function is shown in line 4.
- In line 5 we print the list referenced by `list2`. The output of the `print` function is shown in line 6. Notice that it is the same as the output shown in line 4.
- In line 7 we change the value of `list[0]` to 99.
- In line 8 we print the list referenced by `list1`. The output of the `print` function is shown in line 9. Notice that the first element is now 99.
- In line 10 we print the list referenced by `list2`. The output of the `print` function is shown in line 11. Notice that the first element is 99.

In this interactive session, the `list1` and `list2` variables reference the same list in memory.

Suppose you wish to make a copy of the list, so that `list1` and `list2` reference two separate but identical lists. One way to do this is with a loop that copies each element of the list. Here is an example:

```
# Create a list with values.
list1 = [1, 2, 3, 4]
# Create an empty list.
list2 = []
# Copy the elements of list1 to list2.
for item in list1:
    list2.append(item)
```

After this code executes, `list1` and `list2` will reference two separate but identical lists. A simpler and more elegant way to accomplish the same task is to use the concatenation operator, as shown here:

```
# Create a list with values.
list1 = [1, 2, 3, 4]
# Create a copy of list1.
list2 = [] + list1
```

The last statement in this code concatenates an empty list with `list1` and assigns the resulting list to `list2`. As a result, `list1` and `list2` will reference two separate but identical lists.

7.7 Processing Lists

So far you've learned a wide variety of techniques for working with lists. Now we will look at a number of ways that programs can process the data held in a list. For example, the following *In the Spotlight* section shows how list elements can be used in calculations.

In the Spotlight:

Using List Elements in a Math Expression

Megan owns a small neighborhood coffee shop, and she has six employees who work as baristas (coffee bartenders). All of the employees have the same hourly pay rate. Megan has asked you to design a program that will allow her to enter the number of hours worked by each employee and then display the amounts of all the employees' gross pay. You determine that the program should perform the following steps:

1. For each employee: get the number of hours worked and store it in a list element.
2. For each list element: use the value stored in the element to calculate an employee's gross pay. Display the amount of the gross pay.

Program 7-7 shows the code for the program.

Program 7-7 (barista_pay.py)

```
1  # This program calculates the gross pay for
2  # each of Megan's baristas.
3
4  # NUM_EMPLOYEES is used as a constant for the
5  # size of the list.
6  NUM_EMPLOYEES = 6
7
```

```
 8  def main():
 9    # Create a list to hold employee hours.
10    hours = [0] * NUM_EMPLOYEES
11
12    # Get each employee's hours worked.
13    for index in range(NUM_EMPLOYEES):
14        print('Enter the hours worked by employee ', \
15              index + 1, ': ', sep='', end='')
16        hours[index] = float(input())
17
18    # Get the hourly pay rate.
19    pay_rate = float(input('Enter the hourly pay rate: '))
20
21    # Display each employee's gross pay.
22    for index in range(NUM_EMPLOYEES):
23        gross_pay = hours[index] * pay_rate
24        print('Gross pay for employee ', index + 1, ': $', \
25              format(gross_pay, ',.2f'), sep='')
26
27  # Call the main function.
28  main()
```

Program Output (with input shown in bold)

```
Enter the hours worked by employee 1: 10 Enter
Enter the hours worked by employee 2: 20 Enter
Enter the hours worked by employee 3: 15 Enter
Enter the hours worked by employee 4: 40 Enter
Enter the hours worked by employee 5: 20 Enter
Enter the hours worked by employee 6: 18 Enter
Enter the hourly pay rate: 12.75 Enter
Gross pay for employee 1: $127.50
Gross pay for employee 2: $255.00
Gross pay for employee 3: $191.25
Gross pay for employee 4: $510.00
Gross pay for employee 5: $255.00
Gross pay for employee 6: $229.50
```

NOTE: Suppose Megan's business increases and she hires two additional baristas. This would require you to change the program so it processes eight employees instead of six. Because you used a constant for the list size, this is a simple modification—you just change the statement in line 6 to read:

```
NUM_EMPLOYEES = 8
```

(continued)

Because the NUM_EMPLOYEES constant is used in line 10 to create the list, the size of the hours list will automatically become eight. Also, because you used the NUM_EMPLOYEES constant to control the loop iterations in lines 13 and 22, the loops will automatically iterate eight times, once for each employee.

Imagine how much more difficult this modification would be if you had not used a constant to determine the list size. You would have to change each individual statement in the program that refers to the list size. Not only would this require more work, but it would open the possibility for errors. If you overlooked any one of the statements that refer to the list size, a bug would occur.

Totaling the Values in a List

Assuming a list contains numeric values, to calculate the total of those values you use a loop with an accumulator variable. The loop steps through the list, adding the value of each element to the accumulator. Program 7-8 demonstrates the algorithm with a list named numbers.

Program 7-8 (total_list.py)

```
1  # This program calculates the total of the values
2  # in a list.
3
4  def main():
5      # Create a list.
6      numbers = [2, 4, 6, 8, 10]
7
8      # Create a variable to use as an accumulator.
9      total = 0
10
11     # Calculate the total of the list elements.
12     for value in numbers:
13         total += value
14
15     # Display the total of the list elements.
16     print('The total of the elements is', total)
17
18 # Call the main function.
19 main()
```

Program Output

```
The total of the elements is 30
```

Averaging the Values in a List

The first step in calculating the average of the values in a list is to get the total of the values. You saw how to do that with a loop in the preceding section. The second step is

to divide the total by the number of elements in the list. Program 7-9 demonstrates the algorithm.

Program 7-9 (average_list.py)

```
1   # This program calculates the average of the values
2   # in a list.
3
4   def main():
5       # Create a list.
6       scores = [2.5, 7.3, 6.5, 4.0, 5.2]
7
8       # Create a variable to use as an accumulator.
9       total = 0.0
10
11      # Calculate the total of the list elements.
12      for value in scores:
13          total += value
14
15      # Calculate the average of the elements.
16      average = total / len(scores)
17
18      # Display the total of the list elements.
19      print('The average of the elements is', average)
20
21  # Call the main function.
22  main()
```

Program Output

```
The average of the elements is 5.3
```

Passing a List as an Argument to a Function

Recall from Chapter 5 that as a program grows larger and more complex, it should be broken down into functions that each performs a specific task. This makes the program easier to understand and to maintain.

You can easily pass a list as an argument to a function. This gives you the ability to put many of the operations that you perform on a list in their own functions. When you need to call these functions, you can pass the list as an argument.

Program 7-10 shows an example of a program that uses such a function. The function in this program accepts a list as an argument and returns the total of the list's elements.

Program 7-10 (total_function.py)

```
1  # This program uses a function to calculate the
2  # total of the values in a list.
3
4  def main():
5      # Create a list.
6      numbers = [2, 4, 6, 8, 10]
7
8      # Display the total of the list elements.
9      print('The total is', get_total(numbers))
10
11 # The get_total function accepts a list as an
12 # argument returns the total of the values in
13 # the list.
14 def get_total(value_list):
15     # Create a variable to use as an accumulator.
16     total = 0
17
18     # Calculate the total of the list elements.
19     for num in value_list:
20         total += num
21
22     # Return the total.
23     return total
24
25 # Call the main function.
26 main()
```

Program Output

```
The total is 30
```

Returning a List from a Function

A function can return a reference to a list. This gives you the ability to write a function that creates a list and adds elements to it and then returns a reference to the list so other parts of the program can work with it. The code in Program 7-11 shows an example. It uses a function named `get_values` that gets a series of values from the user, stores them in a list, and then returns a reference to the list.

Program 7-11 (return_list.py)

```
1  # This program uses a function to create a list.
2  # The function returns a reference to the list.
3
```

```
def main():
    # Get a list with values stored in it.
    numbers = get_values()

    # Display the values in the list.
    print('The numbers in the list are:')
    print(numbers)

# The get_values function gets a series of numbers
# from the user and stores them in a list. The
# function returns a reference to the list.
def get_values():
    # Create an empty list.
    values = []

    # Create a variable to control the loop.
    again = 'y'

    # Get values from the user and add them to
    # the list.
    while again == 'y':
        # Get a number and add it to the list.
        num = int(input('Enter a number: '))
        values.append(num)

        # Want to do this again?
        print('Do you want to add another number?')
        again = input('y = yes, anything else = no: ')
        print()

    # Return the list.
    return values

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a number: 1 Enter
Do you want to add another number?
y = yes, anything else = no: y Enter

Enter a number: 2 Enter
Do you want to add another number?
y = yes, anything else = no: y Enter

Enter a number: 3 Enter
Do you want to add another number?
y = yes, anything else = no: y Enter
```

(program output continues)

Program Output *(continued)*

```
Enter a number: 4 Enter
Do you want to add another number?
y = yes, anything else = no: y Enter

Enter a number: 5 Enter
Do you want to add another number?
y = yes, anything else = no: n Enter
The numbers in the list are:
[1, 2, 3, 4, 5]
```

In the Spotlight:

Processing a List

Dr. LaClaire gives a series of exams during the semester in her chemistry class. At the end of the semester she drops each student's lowest test score before averaging the scores. She has asked you to design a program that will read a student's test scores as input and calculate the average with the lowest score dropped. Here is the algorithm that you developed:

Get the student's test scores.
Calculate the total of the scores.
Find the lowest score.
Subtract the lowest score from the total. This gives the adjusted total.
Divide the adjusted total by 1 less than the number of test scores. This is the average.
Display the average.

Program 7-12 shows the code for the program, which is divided into three functions. Rather than presenting the entire program at once, let's first examine the `main` function and then each additional function separately. Here is the `main` function:

Program 7-12 drop_lowest_score.py: `main` function

```
 1  # This program gets a series of test scores and
 2  # calculates the average of the scores with the
 3  # lowest score dropped.
 4
 5  def main():
 6      # Get the test scores from the user.
 7      scores = get_scores()
 8
 9      # Get the total of the test scores.
10      total = get_total(scores)
11
12      # Get the lowest test score.
```

```
13      lowest = min(scores)
14
15      # Subtract the lowest score from the total.
16      total -= lowest
17
18      # Calculate the average. Note that we divide
19      # by 1 less than the number of scores because
20      # the lowest score was dropped.
21      average = total / (len(scores) - 1)
22
23      # Display the average.
24      print('The average, with the lowest score dropped', \
25            'is:', average)
26
```

Line 7 calls the `get_scores` function. The function gets the test scores from the user and returns a reference to a list containing those scores. The list is assigned to the scores variable.

Line 10 calls the `get_total` function, passing the `scores` list as an argument. The function returns the total of the values in the list. This value is assigned to the `total` variable.

Line 13 calls the built-in `min` function, passing the `scores` list as an argument. The function returns the lowest value in the list. This value is assigned to the `lowest` variable.

Line 16 subtracts the lowest test score from the `total` variable. Then, line 21 calculates the average by dividing `total` by `len(scores) - 1`. (The program divides by `len (scores) - 1` because the lowest test score was dropped.) Lines 24 and 25 display the average.

Next is the `get_scores` function.

Program 7-12 drop_lowest_score.py: get_scores function

```
27  # The get_scores function gets a series of test
28  # scores from the user and stores them in a list.
29  # A reference to the list is returned.
30  def get_scores():
31      # Create an empty list.
32      test_scores = []
33
34      # Create a variable to control the loop.
35      again = 'y'
36
37      # Get the scores from the user and add them to
38      # the list.
39      while again == 'y':
40          # Get a score and add it to the list.
```

(program continues)

Program 7-12 *(continued)*

```
41          value = float(input('Enter a test score: '))
42          test_scores.append(value)
43
44          # Want to do this again?
45          print('Do you want to add another score?')
46          again = input('y = yes, anything else = no: ')
47          print()
48
49      # Return the list.
50      return test_scores
51
```

The `get_scores` function prompts the user to enter a series of test scores. As each score is entered it is appended to a list. The list is returned in line 50. Next is the `get_total` function.

Program 7-12 drop_lowest_score.py: get_total function

```
52  # The get_total function accepts a list as an
53  # argument returns the total of the values in
54  # the list.
55  def get_total(value_list):
56      # Create a variable to use as an accumulator.
57      total = 0.0
58
59      # Calculate the total of the list elements.
60      for num in value_list:
61          total += num
62
63      # Return the total.
64      return total
65
66  # Call the main function.
67  main()
```

This function accepts a list as an argument. It uses an accumulator and a loop to calculate the total of the values in the list. Line 64 returns the total.

Program Output (with input shown in bold)

```
Enter a test score: 92 Enter
Do you want to add another score?
Y = yes, anything else = no: y Enter

Enter a test score: 67 Enter
Do you want to add another score?
Y = yes, anything else = no: y Enter
```

```
Enter a test score: 75 Enter
Do you want to add another score?
Y = yes, anything else = no: y Enter

Enter a test score: 88 Enter
Do you want to add another score?
Y = yes, anything else = no: n Enter

The average, with the lowest score dropped is: 85.0
```

Working with Lists and Files

Some tasks may require you to save the contents of a list to a file so the data can be used at a later time. Likewise, some situations may require you to read the data from a file into a list. For example, suppose you have a file that contains a set of values that appear in random order and you want to sort the values. One technique for sorting the values in the file would be to read them into a list, call the list's `sort` method, and then write the values in the list back to the file.

Saving the contents of a list to a file is a straightforward procedure. In fact, Python file objects have a method named `writelines` that writes an entire list to a file. A drawback to the `writelines` method, however, is that it does not automatically write a newline (`'\n'`) at the end of each item. Consequently, each item is written to one long line in the file. Program 7-13 demonstrates the method.

Program 7-13 (writelines.py)

```
1  # This program uses the writelines method to save
2  # a list of strings to a file.
3
4  def main():
5      # Create a list of strings.
6      cities = ['New York', 'Boston', 'Atlanta', 'Dallas']
7
8      # Open a file for writing.
9      outfile = open('cities.txt', 'w')
10
11     # Write the list to the file.
12     outfile.writelines(cities)
13
14     # Close the file.
15     outfile.close()
16
17 # Call the main function.
18 main()
```

After this program executes, the `cities.txt` file will contain the following line:

```
New YorkBostonAtlantaDallas
```

An alternative approach is to use the `for` loop to iterate through the list, writing each element with a terminating newline character. Program 7-14 shows an example.

Program 7-14 (write_list.py)

```
1  # This program saves a list of strings to a file.
2
3  def main():
4      # Create a list of strings.
5      cities = ['New York', 'Boston', 'Atlanta', 'Dallas']
6
7      # Open a file for writing.
8      outfile = open('cities.txt', 'w')
9
10     # Write the list to the file.
11     for item in cities:
12         outfile.write(item + '\n')
13
14     # Close the file.
15     outfile.close()
16
17 # Call the main function.
18 main()
```

After this program executes, the `cities.txt` file will contain the following lines:

```
New York
Boston
Atlanta
Dallas
```

File objects in Python have a method named `readlines` that returns a file's contents as a list of strings. Each line in the file will be an item in the list. The items in the list will include their terminating newline character, which in many cases you will want to strip. Program 7-15 shows an example. The statement in line 8 reads the files contents into a list, and the loop in lines 15 through 17 steps through the list, stripping the `'\n'` character from each element.

Program 7-15 (read_list.py)

```
1  # This program reads a file's contents into a list.
2
3  def main():
4      # Open a file for reading.
```

```
    infile = open('cities.txt', 'r')

    # Read the contents of the file into a list.
    cities = infile.readlines()

    # Close the file.
    infile.close()

    # Strip the \n from each element.
    index = 0
    while index < len(cities):
        cities[index] = cities[index].rstrip('\n')
        index += 1

    # Print the contents of the list.
    print(cities)

# Call the main function.
main()
```

Program Output

```
['New York', 'Boston', 'Atlanta', 'Dallas']
```

Program 7-16 shows another example of how a list can be written to a file. In this example, a list of numbers is written. Notice that in line 12, each item is converted to a string with the `str` function, and then a `'\n'` is concatenated to it.

Program 7-16 (write_number_list.py)

```
# This program saves a list of numbers to a file.

def main():
    # Create a list of numbers.
    numbers = [1, 2, 3, 4, 5, 6, 7]

    # Open a file for writing.
    outfile = open('numberlist.txt', 'w')

    # Write the list to the file.
    for item in numbers:
        outfile.write(str(item) + '\n')

    # Close the file.
    outfile.close()

# Call the main function.
main()
```

When you read numbers from a file into a list, the numbers will have to be converted from strings to a numeric type. Program 7-17 shows an example.

Program 7-17 (read_number_list.py)

```
1  # This program reads numbers from a file into a list.
2
3  def main():
4      # Open a file for reading.
5      infile = open('numberlist.txt', 'r')
6
7      # Read the contents of the file into a list.
8      numbers = infile.readlines()
9
10     # Close the file.
11     infile.close()
12
13     # Convert each element to an int.
14     index = 0
15     while index < len(numbers):
16         numbers[index] = int(numbers[index])
17         index += 1
18
19     # Print the contents of the list.
20     print(numbers)
21
22 # Call the main function.
23 main()
```

Program Output

```
[1, 2, 3, 4, 5, 6, 7]
```

7.8 Two-Dimensional Lists

CONCEPT: A two-dimensional list is a list that has other lists as its elements.

The elements of a list can be virtually anything, including other lists. To demonstrate, look at the following interactive session:

```
1  >>> students = [['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']] Enter
2  >>> print(students) Enter
3  [['Joe', 'Kim'], ['Sam', 'Sue'], ['Kelly', 'Chris']]
4  >>> print(students[0]) Enter
5  ['Joe', 'Kim']
6  >>> print(students[1]) Enter
```

```
 7  ['Sam', 'Sue']
 8  >>> print(students[2]) Enter
 9  ['Kelly', 'Chris']
10  >>>
```

Let's take a closer look at each line.

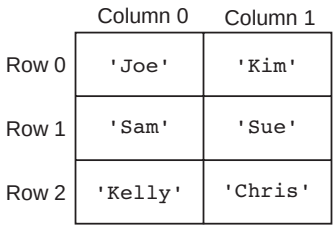
- Line 1 creates a list and assigns it to the `students` variable. The list has three elements, and each element is also a list. The element at `students[0]` is
  `['Joe', 'Kim']`

  The element at `students[1]` is
  `['Sam', 'Sue']`

  The element at `students[2]` is
  `['Kelly', 'Chris']`
- Line 2 prints the entire `students` list. The output of the `print` function is shown in line 3.
- Line 4 prints the `students[0]` element. The output of the `print` function is shown in line 5.
- Line 6 prints the `students[1]` element. The output of the `print` function is shown in line 7.
- Line 8 prints the `students[2]` element. The output of the `print` function is shown in line 9.

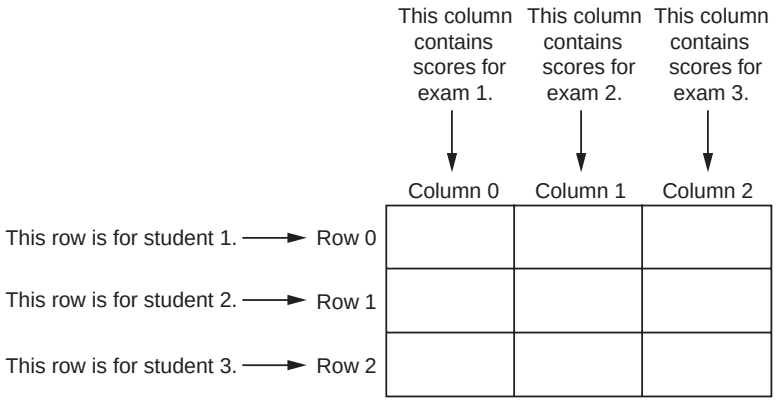
Lists of lists are also known as *nested lists*, or *two-dimensional lists*. It is common to think of a two-dimensional list as having rows and columns of elements, as shown in Figure 7-5. This figure shows the two-dimensional list that was created in the previous interactive session as having three rows and two columns. Notice that the rows are numbered 0, 1, and 2, and the columns are numbered 0 and 1. There is a total of six elements in the list.

Figure 7-5 A two-dimensional list



Two-dimensional lists are useful for working with multiple sets of data. For example, suppose you are writing a grade-averaging program for a teacher. The teacher has three students, and each student takes three exams during the semester. One approach would be to create three separate lists, one for each student. Each of these lists would have three elements, one for each exam score. This approach would be cumbersome, however, because you would have to separately process each of the lists. A better approach would be to use a two-dimensional list with three rows (one for each student) and three columns (one for each exam score), as shown in Figure 7-6.

Figure 7-6 Two-dimensional list with three rows and three columns



When processing the data in a two-dimensional list, you need two subscripts: one for the rows and one for the columns. For example, suppose we create a two-dimensional list with the following statement:

```
scores = [[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]]
```

The elements in row 0 are referenced as follows:

```
scores[0][0]
scores[0][1]
scores[0][2]
```

The elements in row 1 are referenced as follows:

```
scores[1][0]
scores[1][1]
scores[1][2]
```

And, the elements in row 2 are referenced as follows:

```
scores[2][0]
scores[2][1]
scores[2][2]
```

Figure 7-7 illustrates the two-dimensional list, with the subscripts shown for each element.

Figure 7-7 Subscripts for each element of the `scores` list

	Column 0	Column 1	Column 2
Row 0	scores[0][0]	scores[0][1]	scores[0][2]
Row 1	scores[1][0]	scores[1][1]	scores[1][2]
Row 2	scores[2][0]	scores[2][1]	scores[2][2]

Programs that process two-dimensional lists typically do so with nested loops. Let's look at an example. Program 7-18 creates a two-dimensional list and assigns random numbers to each of its elements.

Program 7-18 (random_numbers.py)

```
1  # This program assigns random numbers to
2  # a two-dimensional list.
3  import random
4
5  # Constants for rows and columns
6  ROWS = 3
7  COLS = 4
8
9  def main():
10     # Create a two-dimensional list.
11     values = [[0, 0, 0, 0],
12               [0, 0, 0, 0],
13               [0, 0, 0, 0]]
14
15     # Fill the list with random numbers.
16     for r in range(ROWS):
17         for c in range(COLS):
18             values[r][c] = random.randint(1, 100)
19
20     # Display the random numbers.
21     print(values)
22
23 # Call the main function.
24 main()
```

Program Output

```
[[4, 17, 34, 24], [46, 21, 54, 10], [54, 92, 20, 100]]
```

Let's take a closer look at the program:

- Lines 6 and 7 create global constants for the number of rows and columns.
- Lines 11 through 13 create a two-dimensional list and assign it to the `values` variable. We can think of the list as having three rows and four columns. Each element is assigned the value 0.
- Lines 16 through 18 are a set of nested `for` loops. The outer loop iterates once for each row, and it assigns the variable `r` the values 0 through 2. The inner loop iterates once for each column, and it assigns the variable `c` the values 0 through 3. The statement in line 18 executes once for each element of the list, assigning it a random integer in the range of 1 through 100.
- Line 21 displays the list's contents.

Notice that the statement in line 21 passes the `values` list as an argument to the `print` function; as a result, the entire list is displayed on the screen. Suppose we do not like the way that the `print` function displays the list enclosed in brackets, with each nested list also enclosed in brackets. For example, suppose we want to display each list element on a line by itself, like this:

```
4
17
34
24
46
```

and so forth.

To accomplish that we can write a set of nested loops, such as

```
for r in range(ROWS):
    for c in range(COLS):
        print(values[r][c])
```

Checkpoint

7.19 Look at the following interactive session, in which a two-dimensional list is created. How many rows and how many columns are in the list?

```
numbers = [[1, 2], [10, 20], [100, 200], [1000, 2000]]
```

7.20 Write a statement that creates a two-dimensional list with three rows and four columns. Each element should be assigned the value 0.

7.21 Write a set of nested loops that display the contents of the `numbers` list shown in Checkpoint question 7.19.

7.9 Tuples

CONCEPT: A tuple is an immutable sequence, which means that its contents cannot be changed.

A *tuple* is a sequence, very much like a list. The primary difference between tuples and lists is that tuples are immutable. That means that once a tuple is created, it cannot be changed. When you create a tuple, you enclose its elements in a set of parentheses, as shown in the following interactive session:

```
>>> my_tuple = (1, 2, 3, 4, 5) Enter
>>> print(my_tuple) Enter
(1, 2, 3, 4, 5)
>>>
```

The first statement creates a tuple containing the elements 1, 2, 3, 4, and 5 and assigns it to the variable `my_tuple`. The second statement sends `my_tuple` as an argument to the `print` function, which displays its elements. The following session shows how a `for` loop can iterate over the elements in a tuple:

```
>>> names = ('Holly', 'Warren', 'Ashley') Enter
>>> for n in names: Enter
        print(n) Enter Enter
Holly
Warren
Ashley
>>>
```

Like lists, tuples support indexing, as shown in the following session:

```
>>> names = ('Holly', 'Warren', 'Ashley') Enter
>>> for i in range(len(names)): Enter
        print(names[i]) Enter Enter

Holly
Warren
Ashley
>>>
```

In fact, tuples support all the same operations as lists, except those that change the contents of the list. Tuples support the following:

- Subscript indexing (for retrieving element values only)
- Methods such as `index`
- Built-in functions such as `len`, `min`, and `max`
- Slicing expressions
- The `in` operator
- The `+` and `*` operators

Tuples do not support methods such as `append`, `remove`, `insert`, `reverse`, and `sort`.

NOTE: If you want to create a tuple with just one element, you must write a trailing comma after the element's value, as shown here:

```
my_tuple = (1,)     # Creates a tuple with one element.
```

If you omit the comma, you will not create a tuple. For example, the following statement simply assigns the integer value 1 to the `value` variable:

```
value = (1)         # Creates an integer.
```

What's the Point?

If the only difference between lists and tuples is immutability, you might wonder why tuples exist. One reason that tuples exist is performance. Processing a tuple is faster than processing a list, so tuples are good choices when you are processing lots of data and that data will not be modified. Another reason is that tuples are safe. Because you are not allowed to change the contents of a tuple, you can store data in one and rest assured that it will not be modified (accidentally or otherwise) by any code in your program.

Additionally, there are certain operations in Python that require the use of a tuple. As you learn more about Python, you will encounter tuples more frequently.

Converting Between Lists and Tuples

You can use the built-in `list()` function to convert a tuple to a list and the built-in `tuple()` function to convert a list to a tuple. The following interactive session demonstrates:

```
1  >>> number_tuple = (1, 2, 3) Enter
2  >>> number_list = list(number_tuple) Enter
3  >>> print(number_list) Enter
4  [1, 2, 3]
5  >>> str_list = ['one', 'two', 'three'] Enter
6  >>> str_tuple = tuple(str_list) Enter
7  >>> print(str_tuple) Enter
8  ('one', 'two', 'three')
9  >>>
```

Here's a summary of the statements:

- Line 1 creates a tuple and assigns it to the `number_tuple` variable.
- Line 2 passes `number_tuple` to the `list()` function. The function returns a list containing the same values as `number_tuple`, and it is assigned to the `number_list` variable.
- Line 3 passes `number_list` to the `print` function. The function's output is shown in line 4.
- Line 5 creates a list of strings and assigns it to the `str_list` variable.
- Line 6 passes `str_list` to the `tuple()` function. The function returns a tuple containing the same values as `str_list`, and it is assigned to `str_tuple`.
- Line 7 passes `str_tuple` to the `print` function. The function's output is shown in line 8.

Checkpoint

7.22 What is the primary difference between a list and a tuple?

7.23 Give two reasons why tuples exist.

7.24 Assume that `my_list` references a list. Write a statement that converts it to a tuple.

7.25 Assume that `my_tuple` references a tuple. Write a statement that converts it to a list.