CHAPTER 7

Lists and Tuples

TOPICS

7.1 Sequences

CONCEPT: A sequence is an object that holds multiple items of data, stored one after the other. You can perform operations on a sequence to examine and manipulate the items stored in it.

A *sequence* is an object that contains multiple items of data. The items that are in a sequence are stored one after the other. Python provides various ways to perform operations on the items that are stored in a sequence.

There are several different types of sequence objects in Python. In this chapter, we will look at two of the fundamental sequence types: lists and tuples. Both lists and tuples are sequences that can hold various types of data. The difference between lists and tuples is simple: a list is mutable, which means that a program can change its contents, but a tuple is immutable, which means that once it is created, its contents cannot be changed. We will explore some of the operations that you may perform on these sequences, including ways to access and manipulate their contents.

7.2 Introduction to Lists

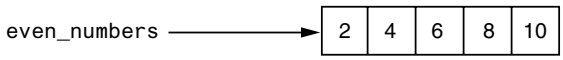
CONCEPT: A list is an object that contains multiple data items. Lists are mutable, which means that their contents can be changed during a program's execution. Lists are dynamic data structures, meaning that items may be added to them or removed from them. You can use indexing, slicing, and various methods to work with lists in a program.

A *list* is an object that contains multiple data items. Each item that is stored in a list is called an *element*. Here is a statement that creates a list of integers:

```
even_numbers = [2, 4, 6, 8, 10]
```

The items that are enclosed in brackets and separated by commas are the list elements. After this statement executes, the variable `even_numbers` will reference the list, as shown in Figure 7-1.

Figure 7-1 A list of integers

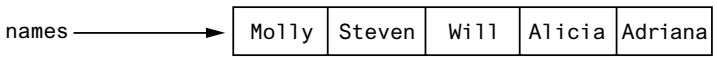


The following is another example:

```
names = ['Molly', 'Steven', 'Will', 'Alicia', 'Adriana']
```

This statement creates a list of five strings. After the statement executes, the `name` variable will reference the list as shown in Figure 7-2.

Figure 7-2 A list of strings

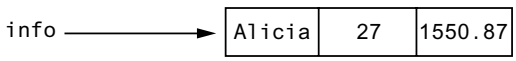


A list can hold items of different types, as shown in the following example:

```
info = ['Alicia', 27, 1550.87]
```

This statement creates a list containing a string, an integer, and a floating-point number. After the statement executes, the `info` variable will reference the list as shown in Figure 7-3.

Figure 7-3 A list holding different types



You can use the `print` function to display an entire list, as shown here:

```
numbers = [5, 10, 15, 20]
print(numbers)
```

In this example, the `print` function will display the elements of the list like this:

```
[5, 10, 15, 20]
```

Python also has a built-in `list()` function that can convert certain types of objects to lists. For example, recall from Chapter 4 that the `range` function returns an iterable, which is an object that holds a series of values that can be iterated over. You can use a statement such as the following to convert the `range` function's iterable object to a list:

```
numbers = list(range(5))
```

When this statement executes, the following things happen:

- The `range` function is called with 5 passed as an argument. The function returns an iterable containing the values 0, 1, 2, 3, 4.
- The `iterable` is passed as an argument to the `list()` function. The `list()` function returns the list `[0, 1, 2, 3, 4]`.
- The list `[0, 1, 2, 3, 4]` is assigned to the `numbers` variable.

Here is another example:

```
numbers = list(range(1, 10, 2))
```

Recall from Chapter 4 that when you pass three arguments to the `range` function, the first argument is the starting value, the second argument is the ending limit, and the third argument is the step value. This statement will assign the list `[1, 3, 5, 7, 9]` to the `numbers` variable.

The Repetition Operator

You learned in Chapter 2 that the `*` symbol multiplies two numbers. However, when the operand on the left side of the `*` symbol is a sequence (such as a list) and the operand on the right side is an integer, it becomes the *repetition operator*. The repetition operator makes multiple copies of a list and joins them all together. Here is the general format:

```
list * n
```

In the general format, *`list`* is a list, and *n* is the number of copies to make. The following interactive session demonstrates:

```
1  >>> numbers = [0] * 5 Enter
2  >>> print(numbers) Enter
3  [0, 0, 0, 0, 0]
4  >>>
```

Let's take a closer look at each statement:

- In line 1, the expression `[0] * 5` makes five copies of the list `[0]` and joins them all together in a single list. The resulting list is assigned to the `numbers` variable.
- In line 2, the `numbers` variable is passed to the `print` function. The function's output is shown in line 3.

Here is another interactive mode demonstration:

```
1  >>> numbers = [1, 2, 3] * 3 Enter
2  >>> print(numbers) Enter
3  [1, 2, 3, 1, 2, 3, 1, 2, 3]
4  >>>
```

NOTE: Most programming languages allow you to create sequence structures known as *arrays*, which are similar to lists, but are much more limited in their capabilities. You cannot create traditional arrays in Python because lists serve the same purpose and provide many more built-in capabilities.

Iterating over a List with the `for` Loop

In Section 7.1, we discussed techniques for accessing the individual characters in a string. Many of the same programming techniques also apply to lists. For example, you can iterate over a list with the `for` loop, as shown here:

```
numbers = [99, 100, 101, 102]
for n in numbers:
    print(n)
```

If we run this code, it will print:

```
99
100
101
102
```

Indexing

Another way that you can access the individual elements in a list is with an *index*. Each element in a list has an index that specifies its position in the list. Indexing starts at 0, so the index of the first element is 0, the index of the second element is 1, and so forth. The index of the last element in a list is 1 less than the number of elements in the list.

For example, the following statement creates a list with 4 elements:

```
my_list = [10, 20, 30, 40]
```

The indexes of the elements in this list are 0, 1, 2, and 3. We can print the elements of the list with the following statement:

```
print(my_list[0], my_list[1], my_list[2], my_list[3])
```

The following loop also prints the elements of the list:

```
index = 0
while index < 4:
    print(my_list[index])
    index += 1
```

You can also use negative indexes with lists to identify element positions relative to the end of the list. The Python interpreter adds negative indexes to the length of the list to determine the element position. The index –1 identifies the last element in a list, –2 identifies the next to last element, and so forth. The following code shows an example:

```
my_list = [10, 20, 30, 40]
print(my_list[-1], my_list[-2], my_list[-3], my_list[-4])
```

In this example, the `print` function will display:

```
40    30    20    10
```

An `IndexError` exception will be raised if you use an invalid index with a list. For example, look at the following code:

```
# This code will cause an IndexError exception.
my_list = [10, 20, 30, 40]
```

```
index = 0
while index < 5:
    print(my_list[index])
    index += 1
```

The last time that this loop begins an iteration, the `index` variable will be assigned the value 4, which is an invalid index for the list. As a result, the statement that calls the `print` function will cause an `IndexError` exception to be raised.

The `len` Function

Python has a built-in function named `len` that returns the length of a sequence, such as a list. The following code demonstrates:

```
my_list = [10, 20, 30, 40]
size = len(my_list)
```

The first statement assigns the list `[10, 20, 30, 40]` to the `my_list` variable. The second statement calls the `len` function, passing the `my_list` variable as an argument.

The function returns the value 4, which is the number of elements in the list. This value is assigned to the `size` variable.

The `len` function can be used to prevent an `IndexError` exception when iterating over a list with a loop. Here is an example:

```
my_list = [10, 20, 30, 40]
index = 0
while index < len(my_list):
    print(my_list[index])
    index += 1
```

Lists Are Mutable

Lists in Python are *mutable*, which means their elements can be changed. Consequently, an expression in the form *`list[index]`* can appear on the left side of an assignment operator. The following code shows an example:

```
1  numbers = [1, 2, 3, 4, 5]
2  print(numbers)
3  numbers[0] = 99
4  print(numbers)
```

The statement in line 2 will display

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

The statement in line 3 assigns 99 to `numbers[0]`. This changes the first value in the list to 99. When the statement in line 4 executes, it will display

```
[99, 2, 3, 4, 5]
```

When you use an indexing expression to assign a value to a list element, you must use a valid index for an existing element or an `IndexError` exception will occur. For example,

look at the following code:

```
numbers = [1, 2, 3, 4, 5]     # Create a list with 5 elements.
numbers[5] = 99               # This raises an exception!
```

The `numbers` list that is created in the first statement has five elements, with the indexes 0 through 4. The second statement will raise an `IndexError` exception because the `numbers` list has no element at index 5.

If you want to use indexing expressions to fill a list with values, you have to create the list first, as shown here:

```
1  # Create a list with 5 elements.
2  numbers = [0] * 5
3
4  # Fill the list with the value 99.
5  index = 0
6  while index < len(numbers):
7    numbers[index] = 99
8    index += 1
```

The statement in line 2 creates a list with five elements, each element assigned the value 0. The loop in lines 6 through 8 then steps through the list elements, assigning 99 to each one.

Program 7-1 shows an example of how user input can be assigned to the elements of a list. This program gets sales amounts from the user and assigns them to a list.

Program 7-1 (sales_list.py)

```
1  # The NUM_DAYS constant holds the number of
2  # days that we will gather sales data for.
3  NUM_DAYS = 5
4
5  def main():
6      # Create a list to hold the sales
7      # for each day.
8      sales = [0] * NUM_DAYS
9
10     # Create a variable to hold an index.
11     index = 0
12
13     print('Enter the sales for each day.')
14
15     # Get the sales for each day.
16     while index < NUM_DAYS:
17         print('Day #', index + 1, ': ', sep='', end='')
18         sales[index] = float(input())
19         index += 1
20
```

```
21      # Display the values entered.
22      print('Here are the values you entered:')
23      for value in sales:
24          print(value)
25
26  # Call the main function.
27  main()
```

Program Output (with input shown in bold)

```
Enter the sales for each day.
Day #1: 1000 Enter
Day #2: 2000 Enter
Day #3: 3000 Enter
Day #4: 4000 Enter
Day #5: 5000 Enter
Here are the values you entered:
1000.0
2000.0
3000.0
4000.0
5000.0
```

The statement in line 3 creates the variable `NUM_DAYS`, which is used as a constant for the number of days. The statement in line 8 creates a list with five elements, with each element assigned the value 0. Line 11 creates a variable named `index` and assigns the value 0 to it.

The loop in lines 16 through 19 iterates 5 times. The first time it iterates, `index` references the value 0, so the statement in line 18 assigns the user's input to `sales[0]`. The second time the loop iterates, index references the value 1, so the statement in line 18 assigns the user's input to `sales[1]`. This continues until input values have been assigned to all the elements in the list.

Concatenating Lists

To concatenate means to join two things together. You can use the + operator to concatenate two lists. Here is an example:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list3 = list1 + list2
```

After this code executes, `list1` and `list2` remain unchanged, and `list3` references the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

The following interactive mode session also demonstrates list concatenation:

```
>>> girl_names = ['Joanne', 'Karen', 'Lori'] Enter
>>> boy_names = ['Chris', 'Jerry', 'Will'] Enter
>>> all_names = girl_names + boy_names Enter
```

```
>>> print(all_names) Enter
['Joanne', 'Karen', 'Lori', 'Chris', 'Jerry', 'Will']
```

You can also use the += augmented assignment operator to concatenate one list to another. Here is an example:

```
list1 = [1, 2, 3, 4]
list2 = [5, 6, 7, 8]
list1 += list2
```

The last statement appends `list2` to `list1`. After this code executes, `list2` remains unchanged, but `list1` references the following list:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

The following interactive mode session also demonstrates the += operator used for list concatenation:

```
>>> girl_names = ['Joanne', 'Karen', 'Lori'] Enter
>>> girl_names += ['Jenny', 'Kelly'] Enter
>>> print(girl_names) Enter
['Joanne', 'Karen', 'Lori', 'Jenny', 'Kelly']
>>>
```

NOTE: Keep in mind that you can concatenate lists only with other lists. If you try to concatenate a list with something that is not a list, an exception will be raised.

Checkpoint

7.1 What will the following code display?

```
numbers = [1, 2, 3, 4, 5]
numbers[2] = 99
print(numbers)
```

7.2 What will the following code display?

```
numbers = list(range(3))
print(numbers)
```

7.3 What will the following code display?

```
numbers = [10] * 5
print(numbers)
```

7.4 What will the following code display?

```
numbers = list(range(1, 10, 2))
for n in numbers:
    print(n)
```

7.5 What will the following code display?

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

7.6 How do you find the number of elements in a list?

7.7 What will the following code display?

```
numbers1 = [1, 2, 3]
numbers2 = [10, 20, 30]
numbers3 = numbers1 + numbers2
print(numbers1)
print(numbers2)
print(numbers3)
```

7.8 What will the following code display?

```
numbers1 = [1, 2, 3]
numbers2 = [10, 20, 30]
numbers2 += numbers1
print(numbers1)
print(numbers2)
```

7.3 List Slicing

CONCEPT: A slicing expression selects a range of elements from a sequence.

You have seen how indexing allows you to select a specific element in a sequence. Sometimes you want to select more than one element from a sequence. In Python, you can write expressions that select subsections of a sequence, known as slices.


VideoNote
List Slicing

A *slice* is a span of items that are taken from a sequence. When you take a slice from a list, you get a span of elements from within the list. To get a slice of a list, you write an expression in the following general format:

```
list_name[start : end]
```

In the general format, *start* is the index of the first element in the slice, and *end* is the index marking the end of the slice. The expression returns a list containing a copy of the elements from *start* up to (but not including) *end*. For example, suppose we create the following list:

```
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday', 'Saturday']
```

The following statement uses a slicing expression to get the elements from indexes 2 up to, but not including, *5*:

```
mid_days = days[2:5]
```

After this statement executes, the `mid_days` variable references the following list:

```
['Tuesday', 'Wednesday', 'Thursday']
```

You can quickly use the interactive mode interpreter to see how slicing works. For example, look at the following session. (We have added line numbers for easier reference.)

```
1  >>> numbers = [1, 2, 3, 4, 5] Enter
2  >>> print(numbers) Enter
3  [1, 2, 3, 4, 5]
```

```
4  >>> print(numbers[1:3]) Enter
5  [2, 3]
6  >>>
```

Here is a summary of each line:

- In line 1, we created the list and `[1, 2, 3, 4, 5]` and assigned it to the `numbers` variable.
- In line 2, we passed `numbers` as an argument to the `print` function. The `print` function displayed the list in line 3.
- In line 4, we sent the slice `numbers[1:3]` as an argument to the `print` function. The `print` function displayed the slice in line 5.

If you leave out the *`start`* index in a slicing expression, Python uses 0 as the starting index. The following interactive mode session shows an example:

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

Notice line 4 sends the slice `numbers[:3]` as an argument to the `print` function. Because the starting index was omitted, the slice contains the elements from index 0 up to 3.

If you leave out the *`end`* index in a slicing expression, Python uses the length of the list as the *`end`* index. The following interactive mode session shows an example:

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

Notice line 4 sends the slice `numbers[2:]` as an argument to the `print` function. Because the ending index was omitted, the slice contains the elements from index 2 through the end of the list.

If you leave out both the *`start`* and *`end`* index in a slicing expression, you get a copy of the entire list. The following interactive mode session shows an example:

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

The slicing examples we have seen so far get slices of consecutive elements from lists. Slicing expressions can also have step value, which can cause elements to be skipped in the list. The following interactive mode session shows an example of a slicing expression with a step value:

```
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 you saw earlier removes a specific item from a list, if that item is in the list. Some situations might require 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

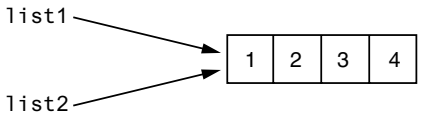
CONCEPT: To make a copy of a list, you must copy the list's elements.

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 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 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 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 `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, then display the amounts of all the employees' gross pay. You determine 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
```