In many programs, you need to collect large numbers of values. In Python, you use the list structure for this purpose. A list is a container that stores a collection of elements that are arranged in a linear or sequential order. Lists can automatically grow to any desired size as new items are added and shrink as items are removed. In this chapter, you will learn about lists and several common algorithms for processing them.

6.1 Basic Properties of Lists

We start this chapter by introducing the `list` data type. Lists are the fundamental mechanism in Python for collecting multiple values. In the following sections, you will learn how to create lists and how to access list elements.

6.1.1 Creating Lists

Suppose you write a program that reads a sequence of values and prints out the sequence, marking the largest value, like this:

```
32
54
67.5
29
35
80
115 <= largest value
44.5
100
65
```

You do not know which value to mark as the largest one until you have seen them all. After all, the last value might be the largest one. Therefore, the program must first store all values before it can print them.

Could you simply store each value in a separate variable? If you know that there are ten values, then you could store the values in ten variables `value1`, `value2`, `value3`, ..., `value10`. However, such a sequence of variables is not very practical to use. You would have to write quite a bit of code ten times, once for each of the variables. In Python, a **list** is a much better choice for storing a sequence of values.

A list is a container that stores a sequence of values.

Here we create a list and specify the initial values that are to be stored in the new list (see Figure 1):

```
values = [32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65]
```

(1)

The square brackets indicate that we are creating a list. The items are stored in the order they are provided. You will want to store the list in a variable so that you can access it later.

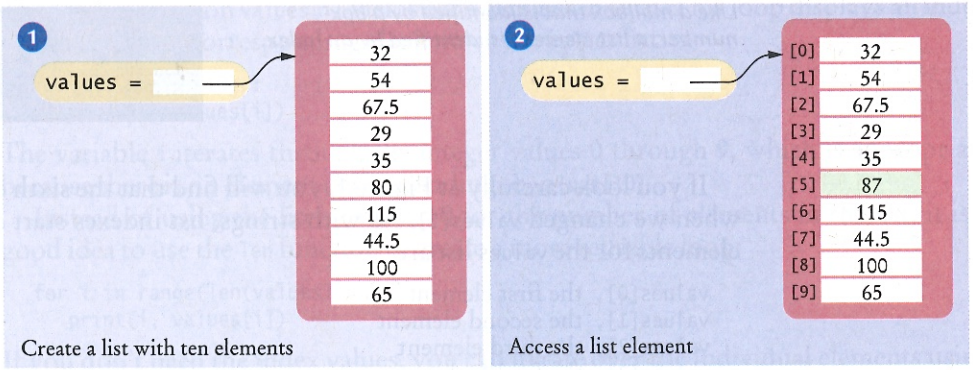


Figure 1 A List of Size 10

6.1.2 Accessing List Elements

Each individual element in a list is accessed by an integer i, using the notation *list*[i].

A list is a sequence of *elements*, each of which has an integer position or *index*. To access a list element, you specify which index you want to use. That is done with the subscript operator ([]) in the same way that you access individual characters in a string. For example,

```
print(values[5])    # Prints the element at index 5
```

This is not an accident. Both lists and strings are **sequences**, and the [] operator can be used to access an element in any sequence.

There are two differences between lists and strings. Lists can hold values of any type, whereas strings are sequences of characters. Moreover, strings are immutable—you cannot change the characters in the sequence. But lists are *mutable*. You can replace one list element with another, like this:

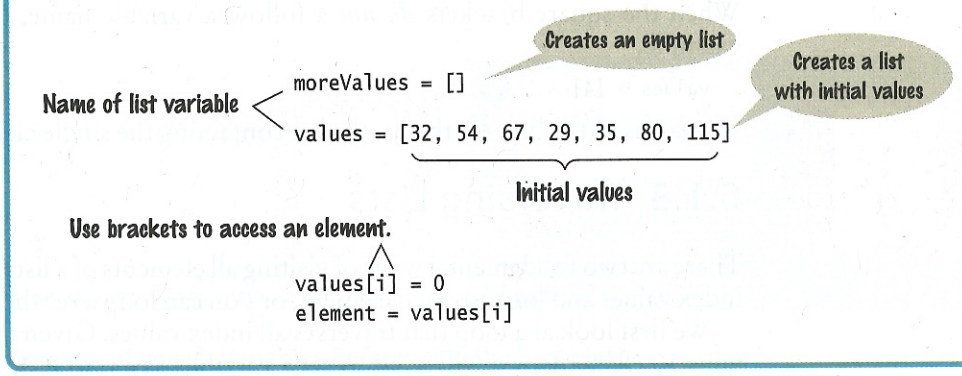
```
values[5] = 87
```

2

Now the element at index 5 is filled with 87 (see Figure 1).

Syntax 6.1 Lists

Syntax	To create a list:	[*value*$_1$, *value*$_2$, . . .]
	To access an element:	*listReference*[*index*]



Like a mailbox that is identified by a box number, a list element is identified by an index.

If you look carefully at Figure 1, you will find that the sixth element was modified when we changed `values[5]`. As with strings, list indexes start at 0. That is, the legal elements for the `values` list are

```
values[0], the first element
values[1], the second element
values[2], the third element
values[3], the fourth element
values[4], the fifth element
. . .
values[9], the tenth element
```

In this list, an index can be any integer ranging from 0 to 9.

A list index must be at least zero and less than the number of elements in the list.

You have to be careful that the index stays within the valid range. Trying to access an element that does not exist in the list is a serious error. For example, if `values` has ten elements, you are not allowed to access `values[20]`. Attempting to access an element whose index is not within the valid index range is called an **out-of-range error** or a bounds error. When an out-of-range error occurs at run time, it causes a run-time exception.

An out of range error, which occurs if you supply an invalid list index, can cause your program to terminate.

Here is a very common bounds error:

```
values[10] = number
```

There is no `values[10]` in a list with ten elements – the index can range from 0 to 9. To avoid out-of-range errors, you will want to know how many elements are in a list. You can use the `len` function to obtain the *length* of the list; that is, the number of elements:

```
numElements = len(values)
```

The following code ensures that you only access the list when the index variable `i` is within the legal bounds:

```
if 0 <= i and i < len(values) :
   values[i] = number
```

Note that there are two distinct uses of the square brackets. When the square brackets immediately follow a variable name, they are treated as the subscript operator, as in

```
values[4]
```

When the square brackets *do not* a follow a variable name, they create a list. For example,

```
values = [4]
```

sets `values` to the list `[4]`; that is, the list containing the single element 4.

6.1.3 Traversing Lists

There are two fundamental ways of visiting all elements of a list. You can loop over the index values and look up each element, or you can loop over the elements themselves.

We first look at a loop that traverses all index values. Given the `values` list that contains 10 elements, we will want to set a variable, say `i`, to 0, 1, 2, and so on, up to 9.

Then the expression `values[i]` yields each element in turn. This loop displays all index values and their corresponding elements in the `values` list.

```
for i in range(10) :
   print(i, values[i])
```

The variable `i` iterates through the integer values 0 through 9, which is appropriate because there is no element corresponding to `values[10]`.

Instead of using the literal value 10 for the number of elements in the list, it is a good idea to use the `len` function to create a more reusable loop:

```
for i in range(len(values)) :
   print(i, values[i])
```

You can iterate over the index values or the elements of a list.

If you don't need the index values, you can iterate over the individual elements using a `for` loop in the form:

```
for element in values :
   print(element)
```

Note again the similarity between strings and lists. As was the case with looping over the characters in a string, the loop body is executed once for each element in the list `values`. At the beginning of each loop iteration, the next element is assigned to the loop variable `element` and the loop body is then executed.

6.1.4 List References

If you look closely at Figure 1, you will note that the variable `values` does not store any numbers. Instead, the list is stored elsewhere and the `values` variable holds a **reference** to the list. (The reference denotes the location of the list in memory.) When you access the elements in a list, you need not be concerned about the fact that Python uses list references. This only becomes important when copying list references.

A list reference specifies the location of a list. Copying the reference yields a second reference to the same list.

When you copy a list variable into another, both variables refer to the same list (see Figure 2). The second variable is an *alias* for the first because both variables reference the same list.

```
scores = [10, 9, 7, 4, 5]
values = scores   # Copying list reference (1)
```

You can modify the list through either of the variables:

```
scores[3] = 10
print(values[3])   # Prints 10 (2)
```

Section 6.2.8 shows how you can make a copy of the *contents* of the list.

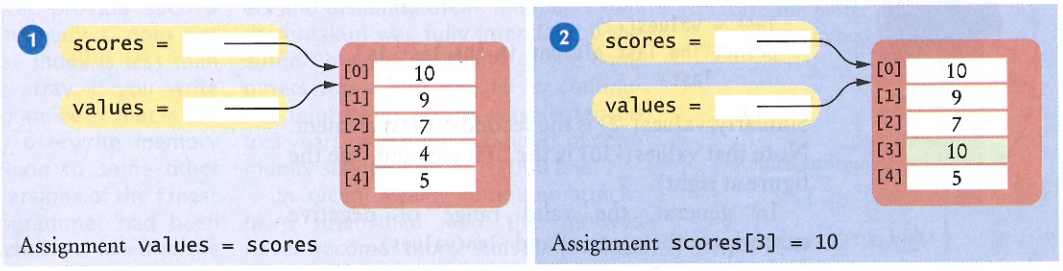


Figure 2 Two List Variables Referencing the Same List

1. Define a list of integers containing the first five prime numbers.
2. Assume that the list `primes` has been initialized as described in Self Check 1. What does it contain after executing the following loop?

```
for i in range(2) :
   primes[4 - i] = primes[i]
```

3. Assume that the list `primes` has been initialized as described in Self Check 1. What does it contain after executing the following loop?

```
for i in range(5) :
   primes[i] = primes[i] + 1
```

4. Given the definition

```
values = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

write statements to put the integer 10 into the elements of the list `values` with the lowest and the highest valid index.

5. Define a list containing two strings, `"Yes"`, and `"No"`.
6. Can you produce the output on page 278 without storing the inputs in a list, by using an algorithm similar to the algorithm for finding the maximum in Section 4.5.4?

Practice It Now you can try these exercises at the end of the chapter: R6.1, R6.2, R6.7, P6.1.

Common Error 6.1

Out-of-Range Errors

Perhaps the most common error in using lists is accessing a nonexistent element.

```
values = [2.3, 4.5, 7.2, 1.0, 12.2, 9.0, 15.2, 0.5]
values[8] = 5.4
   # Error—values has 8 elements, and the index can range from 0 to 7
```

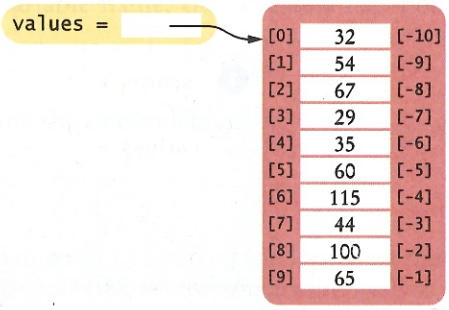
If your program accesses a list through an out-of-range index, the program will generate an exception at run time.

Special Topic 6.1

Reverse Subscripts

Python, unlike many other languages, also allows you to use negative subscripts when accessing an element of a list. The negative subscripts provide access to the list elements in reverse order. For example, a subscript of –1 provides access to the last element in the list:

```
last = values[-1]
print("The last element in the list is",
      last)
```



Similarly, `values[-2]` is the second-to-last element. Note that `values[-10]` is the first element (see the figure at right).

In general, the valid range of negative subscripts is between `-1` and `-len(values)`.

Programming Tip 6.1

Use Lists for Sequences of Related Items

Lists are intended for storing sequences of values with the same meaning. For example, a list of test scores makes perfect sense:

```
scores = [98, 85, 100, 89, 73, 92, 83, 65, 79, 80]
```

But a list

```
personalData = ["John Q. Public", 25, 485.25, "10 wide"]
```

that holds a person's name, age, bank balance, and shoe size in positions 0, 1, 2, and 3 is bad design. It would be tedious for the programmer to remember which of these data values is stored in which list location. In this situation, it is far better to use three separate variables.

Computing & Society 6.1 Computer Viruses

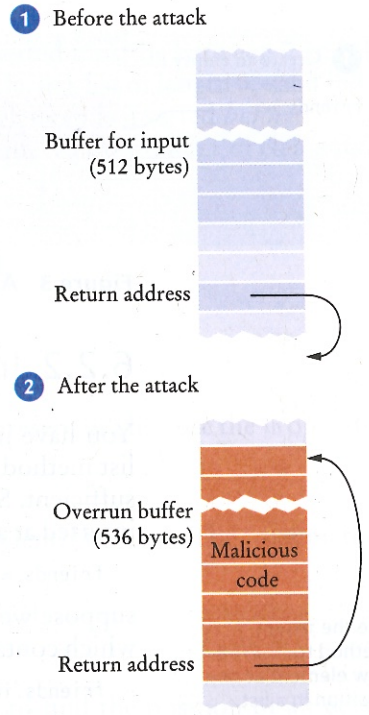
In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected about 6,000 computers connected to the Internet across the United States. Tens of thousands of computer users were unable to read their e-mail or otherwise use their computers. All major universities and many high-tech companies were affected. (The Internet was much smaller then than it is now.)

The particular kind of virus used in this attack is called a *worm*. The worm program crawled from one computer on the Internet to the next. The worm would attempt to connect to `finger`, a program in the UNIX operating system for finding information on a user who has an account on a particular computer on the network. Like many programs in UNIX, `finger` was written in the C language. In order to store the user name, the `finger` program allocated an array of 512 characters (an array is a sequence structure similar to a list), under the assumption that nobody would ever provide such a long input. Unfortunately, C does not check that an array index is less than the length of the array. If you write into an array using an index that is too large, you simply overwrite memory locations that belong to some other objects. In some versions of the `finger` program, the programmer had been lazy and had not checked whether the array holding the input characters was large enough to hold the input. So the worm program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes would overwrite a return address, which the attacker knew was stored just after the array. When that method was finished, it didn't return to its caller but to code supplied by the worm (see the figure, A "Buffer Overrun" Attack). That code ran under the same super-user privileges as `finger`, allowing the worm to gain entry into the remote system. Had the programmer who wrote `finger` been more conscientious, this particular attack would not be possible.

In Python, as in C, all programmers must be very careful not to overrun the boundaries of a sequence. However, in Python, this error causes a run-time exception and never corrupts memory outside the list.

One may well speculate what would possess the virus author to spend many weeks to plan the antisocial act of breaking into thousands of computers and disabling them. It appears that the break-in was fully intended by the author, but the disabling of the computers was a bug, caused by continuous reinfection. Morris was sentenced to 3 years probation, 400 hours of community service, and a $10,000 fine.

In recent years, computer attacks have intensified and the motives have become more sinister. Instead of disabling computers, viruses often steal financial data or use the attacked computers for sending spam e-mail. Sadly, many of these attacks continue to be possible because of poorly written programs that are susceptible to buffer overrun errors.



A "Buffer Overrun" Attack