GLOBAL EDITION

Introduction to Python® Programming and Data Structures

THIRD EDITION

Y. Daniel Liang

Digital Resources for Students

Your new eBook provides 12-month access to digital resources that include VideoNotes, animations, case studies, supplements, and more on the Companion Website. Refer to the preface in the textbook for a detailed list of resources.

Follow these instructions to register for the Companion Website:

1. Go to www.pearsonglobaleditions.com.
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

ISSLIA-SHELL-SHONE-SNOBS-WERSH-UNRWA

Use the login name and password you created during registration to start using the online resources that accompany your textbook.

IMPORTANT:

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable.

For technical support, go to https://support.pearson.com/getsupport

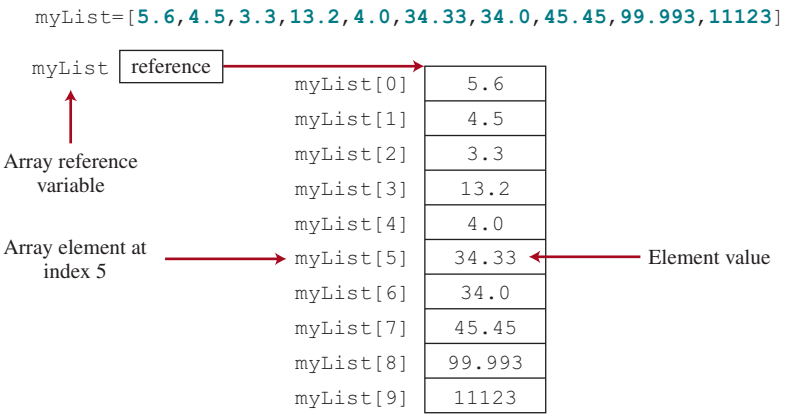

FIGURE 7.1 The list `myList` has 10 elements with indexes from `0` to `9`.

Caution

Accessing a list out of bounds is a common programming error that results in a runtime `IndexError`. To avoid this error, make sure that you do not use an index beyond `len(myList) - 1`.

Programmers often mistakenly reference the first element in a list with index `1`, but it should be `0`. This is called the *off-by-one error*. It is a common error in a loop to use `<=` where `<` should be used. For example, the following loop is wrong:

```
i = 0
while i <= len(myList):
    print(myList[i])
    i += 1
```

The `<=` should be replaced by `<`.

Python also allows the use of negative numbers as indexes to reference positions relative to the end of the list. The actual position is obtained by adding the length of the list with the negative index. For example:

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[-1]
3  21
4  >>> list1[-3]
5  2
6  >>>
```

In line 2, `list1[-1]` is same as `list1[-1 + len(list1)]`, which gives the last element in the list. In line 4, `list1[-3]` is same as `list1[-3 + len(list1)]`, which gives the third last element in the list.

7.2.4 List Slicing `[start : end : step]`

The index operator allows you to select an element at the specified index. The slicing operator returns a slice of the list using the syntax `list[start : end : step]`. The slice is a sublist from index `start` to index `end - 1` with the specified `step`. By default, `step` is `1`. Here are some examples:

```
1  >>> list1 = [2, 3, 5, 7, 9, 1]
2  >>> list1[2 : 4]
3  [5, 7]
```

```
>>> list1[0 : 5 : 2]
[2, 5, 9]
>>>
```

The starting index or ending index may be omitted. In this case, the starting index is **0** and the ending index is the last index. For example:

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[ : 2]
3  [2, 3]
4  >>> list1[3 : ]
5  [2, 33, 21]
6  >>>
```

Note that **list1[: 2]** is the same as **list1[0 : 2]** (line 2) and that **list1[3 :]** is the same as **list1[3 : len(list1)]** (line 4).

You can use a negative index in slicing. For example:

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[1 : -3]
3  [3, 5]
4  >>> list1[-4 : -2]
5  [5, 2]
6  >>>
```

In line 2, **list1[1 : -3]** is the same as **list1[1 : -3 + len(list1)]**. In line 4, **list1[-4 : -2]** is the same as **list1[-4 + len(list1) : -2 + len(list1)]**.

You can assign values to a slice of list. For example:

```
1  >>> list1 = [2, 3, 5, 2, 33, 21]
2  >>> list1[1 : 3] = [91, 92, 93, 94]
3  >>> list1
4  [2, 91, 92, 93, 94, 2, 33, 21]
5  >>>
```

In line 2, **list1[1 : 3] = [91, 92, 93, 94]** replaces **[3, 5]** in **list1** with **[91, 92, 93, 94]**.

Note

If **start >= end**, **list[start : end]** returns an empty list. If **end** specifies a position beyond the end of the list, Python will use the length of the list for **end** instead.

7.2.5 The +, +=, *, and in/not in Operators

You can use the concatenation operator (**+**) to join two lists and the repetition operator (*****) to replicate elements in a list. Here are some examples:

```
1  >>> list1 = [2, 3]
2  >>> list2 = [1, 9]
3  >>> list3 = list1 + list2
4  >>> list3
5  [2, 3, 1, 9]
6  >>>
7  >>> list4 = 3 * list1
8  >>> list4
9  [2, 3, 2, 3, 2, 3]
10 >>> list4 += [7, 8]
11 >>> list4
12 [2, 3, 2, 3, 2, 3, 7, 8]
13 >>>
```

A new list is obtained by concatenating `list1` with `list2` (line 3). Line 7 duplicates `list1` three times to create a new list. Note that `3 * list1` is the same as `list1 * 3`. Line 10 appends `[7, 8]` to `list4`.

You can determine whether an element is in a list by using the **in** or **not in** operator. For example:

```
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> 2 in list1
True
>>> 2 not in list1
False
>>>
```

7.2.6 Traversing Elements in a For Loop

The elements in a Python list are iterable. Python supports a convenient `for` loop, which enables you to traverse the list sequentially without using an index variable. For example, the following code displays all the elements in the list `myList`:

```
for u in myList:
    print(u)
```

You can read the code as, "For each element `u` in `myList`, print it."

You still have to use an index variable if you wish to traverse the list in a different order or change the elements in the list. For example, the following code displays the elements at even-numbered indices.

```
for i in range(0, len(myList), 2):
    print(myList[i])
```

7.2.7 Comparing Lists

You can compare lists using the comparison operators (`>`, `>=`, `<`, `<=`, `==`, and `!=`). For this to work, the two lists must contain the same type of elements. The comparison uses *lexicographical* ordering: the first two elements are compared and if they differ, this determines the outcome of the comparison; if they are equal, the next two elements are compared, and so on, until either list is exhausted. Here are some examples:

```
>>> list1 = ["green", "red", "blue"]
>>> list2 = ["red", "blue", "green"]
>>> list2 == list1
False
>>> list2 != list1
True
>>> list2 >= list1
True
>>> list2 > list1
True
>>> list2 < list1
False
>>> list2 <= list1
False
>>>
```

7.2.8 List Comprehensions

List comprehensions is a concise syntax that creates a list by processing another sequence of data. A list comprehension consists of brackets containing an expression followed by a `for`

clause then zero or more `for` or `if` clauses. The list comprehension produces a list with the results from evaluating the expression. Here are some examples:

```
>>> list1 = [x for x in range(5)] # Returns a list [0, 1, 2, 3, 4]
>>> list1
[0, 1, 2, 3, 4]
>>>
>>> list2 = [0.5 * x for x in list1]
>>> list2
[0.0, 0.5, 1.0, 1.5, 2.0]
>>>
>>> list3 = [x for x in list2 if x < 1.5]
>>> list3
[0.0, 0.5, 1.0]
>>>
```

In line 1, `list1` is created from an expression using a `for` clause. The numbers in `list1` are `0`, `1`, `2`, `3`, and `4`. Each number in `list2` is half of the corresponding number in `list1` (line 5). In line 9, `list3` consists of the numbers whose value is less than `1.5` in `list2`.

7.2.9 List Methods

Lists are defined using the `list` class in Python. Once a list is created, you can use the `list` class's methods (shown in Table 7.2) to manipulate the list.

TABLE 7.2 The list Class Contains Methods for Manipulating a List

Method	*Description*
`append(x)`	Adds an element `x` to the end of the list.
`count(x)`	Returns the number of times element `x` appears in the list.
`extend(anotherList)`	Appends all the elements in `anotherList` to the list.
`index(x)`	Returns the index of the first occurrence of element `x` in the list.
`insert(index, x)`	Inserts an element `x` at a given `index`. Note that the first element in the list has index `0`.
`pop(index)`	Removes the element at the given `index` and return it. The parameter index is optional. If it is not specified, `list.pop()` removes and returns the last element in the list.
`remove(x)`	Removes the first occurrence of element `x` from the list.
`reverse()`	Reverses the elements in the list.
`sort()`	Sorts the elements in the list in an ascending order.

Here are some examples that use the `append`, `count`, `extend`, `index`, and `insert` methods:

```
>>> list1 = [2, 3, 4, 1, 32, 4]
>>> list1.append(19)
>>> list1
[2, 3, 4, 1, 32, 4, 19]
>>> list1.count(4) # Return the count for number 4
2
>>> list2 = [99, 54]
>>> list1.extend(list2)
>>> list1
[2, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.index(4) # Return the index of number 4
2
>>> list1.insert(1, 25) # Insert 25 at position index 1
>>> list1
[2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>>
```

Line 2 appends 19 to the list. Line 5 returns the count of the number of occurrences of element 4 in the list. Line 8 appends list2 to list1. Line 11 returns the index for element 4 in the list. Line 13 inserts 25 into the list at index 1.

Here are some examples that use the insert, pop, remove, reverse, and sort methods:

```
>>> list1 = [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop(2)
3
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop()
54
>>> list1
[2, 25, 4, 1, 32, 4, 19, 99]
>>> list1.remove(32) # Remove number 32
>>> list1
[2, 25, 4, 1, 4, 19, 99]
>>> list1.reverse() # Reverse the list
>>> list1
[99, 19, 4, 1, 4, 25, 2]
>>> list1.sort() # Sort the list
>>> list1
[1, 2, 4, 4, 19, 25, 99]
>>> list1.sort(reverse = True) # Sort the list in descending order
>>> list1
[99, 25, 19, 4, 4, 2, 1]
>>>
```

Line 2 removes the element at index 2 from the list. Invoking list1.pop() (line 6) returns and removes the last element from list1. Line 10 removes element 32 from list1. Line 13 reverses the elements in the list. Line 16 sorts the elements in the list in ascending order and line 19 sorts the elements in the list in descending order.

Note

list is a predefined class in Python. To avoid errors, don't name your list using list. This book names a list using the names such as list1, list2, lst, etc.

Note

In many other programming languages, you would use a type called an *array* to store a sequence of data. An array has a fixed size. A Python list's size is flexible. It can grow and shrink on demand.

7.2.10 Splitting a String into a List

To split the characters in a string s into a list, use list(s). For example, list("abc") is ['a', 'b', 'c'].

The str class contains the split method, which is useful for splitting items in a string into a list. For example, the following statement:

```
items = "Areebah Ashley Gabriel Helena".split()
```

splits the string "Areebah Ashley Gabriel Helena" into the list ['Areebah, 'Ashley', 'Gabriel', 'Helena']. In this case the items are delimited by spaces in the string. You can use a nonspace delimiter. For example, the following statement:

```
items = "12/25/1997".split("/")
```

splits the string 12/25/1997 into the list ['12', '25', '1997'].

Note

Python supports *regular expressions*, an extremely useful and powerful feature for matching and splitting a string using a pattern. Regular expressions are complex for beginning students. For this reason, we cover them in Appendix E, "Regular Expressions."

7.2.11 Inputting Lists

To read data from the console into a list, you can enter one data item per line and append it to a list in a loop. For example, the following code reads ten numbers *one per line* into a list.

```
list1 = [] # Create a list
print("Enter 10 numbers, one number per line: ")
for i in range(10):
    list1.append(float(input()))
```

Sometimes it is more convenient to enter the data in one line separated by spaces. You can use the string's `split` method to extract data from a line of input. For example, the following code reads ten numbers separated by spaces on one line into a list.

```
# Read numbers as a string from the console
s = input("Enter 10 numbers separated by spaces on one line: ")
items = s.split() # Extract items from the string
list1 = [float(x) for x in items] # Convert items to numbers
```

Invoking `input()` reads a string. Using `s.split()` extracts the items delimited by spaces from string `s` and returns items in a list. The last line creates a list of numbers by converting the items into numbers.

7.2.12 Shifting Lists

Sometimes you need to shift the elements left or right. Python does not provide such a method in the `list` class, but you can write the following code to perform a left shift.

```
lst = [4, 5, 6, 7, 8, 9]
temp = lst[0] # Retain the first element

# Shift elements left
for i in range(1, len(lst)):
    lst[i - 1] = lst[i]

# Move the first element to fill in the last position
lst[len(lst) - 1] = temp
```

The preceding code can be simplified as follows:

```
lst = lst[1 : len(lst)] + lst[0 : 1]
```

7.2.13 Simplifying Coding Using Lists

Lists can be used to greatly simplify coding for certain tasks. For example, suppose you wish to obtain the English month name for a given month in number. If the month names are stored in a list, the month name for a given month can be accessed simply via index.