Chapter 7

More on **for**-Loops, Lists, and Iterables

The previous chapter introduced `lists`, `tuples`, the `range()` function, and `for`-loops. The reason for introducing these concepts in the same chapter is because either they are closely related (as is true with `lists` and `tuples`) or they are often used together (as is true, for example, with the `range()` function and `for`-loops). In this chapter we want to extend our understanding of the ways in which `for`-loops and iterables can be used. Although the material in this chapter is often presented in terms of `lists`, you should keep in mind that the discussion almost always pertains to `tuples` too—you could substitute a `tuple` for a `list` in the given code and the result would be the same. (This is not true only when it comes to code that assigns values to individual elements. Recall that `lists` are mutable but `tuples` are not. Hence, once a `tuple` is created, we cannot change its elements.)

The previous chapter mentioned that a `list` can have elements that are themselves `lists`, but no details were provided. In this chapter we will dive into some of these details. We will also consider nested `for`-loops, two new ways of indexing (specifically *negative indexing* and *slicing*), and the use of strings as *sequences* or iterables. We start by considering nested `for`-loops.

7.1 `for`-Loops within `for`-Loops

There are many algorithms that require that one loop be *nested* inside another. For example, nested loops can be used to generate data for a table where the inner loop dictates the column and the outer loop dictates the row. In fact, it is not uncommon for algorithms to require several levels of nesting (i.e., a loop within a loop within a loop and so on). As you will see, nested loops are also useful for processing data organized in the form of `lists` within a `list`. We start this section by showing some of the general ways in which `for`-loops can be nested. This is presented primarily in terms of generating various patterns of characters. After introducing `lists` of `lists` (in Sec. 7.2) we will consider more practical applications for nested `for`-loops.

Assume we want to generate the following output:

```
1
12
123
```

From the file: `more-on-iterables.tex`

```
1234
12345
123456
1234567
```

There are seven lines. The first line consists of a single 1. Each successive line has one more digit than the previous line. This output can be generated using *nested* for-loops. Nested for-loops consist of an "outer" for-loop and an "inner" for-loop. The body of the outer for-loop is executed the number of times dictated by its header. Of course, the number of times the body of the inner loop is executed is also dictated by its header. However, the contents of the inner-loop's header can change with each pass of the outer loop.

Turning our attention back to the collection of characters above, we can use the outer loop to specify that we want to generate seven lines of output, i.e., the body of the outer loop will be executed seven times. We can then use the inner loop to generate the characters on each individual line.

Listing 7.1 shows nested for-loops that produce the arrangement of characters shown above.

Listing 7.1 Nested for-loops that generate seven lines of integers.

```
1  >>> for i in range(7):  # Header for outer loop.
2  ...     for j in range(1, i + 2): # Cycle through integers.
3  ...         print(j, end="")      # Suppress newline.
4  ...     print()                   # Add newline.
5  ...
6  1
7  12
8  123
9  1234
10 12345
11 123456
12 1234567
```

Line 1 contains the header of the outer for-loop. The body of this loop executes seven times because the argument of the range() function is 7. Thus, the loop variable i takes on values 0 through 6. The header for the inner for-loop is on line 2. This header contains range(1, i + 2). Notice that the inner loop variable is j. Given the header of the inner loop, j will take on values between 1 and i + 1, inclusive. So, for example, when i is 1, corresponding to the *second* line of output, j varies between 1 and 2. When i is 2, corresponding to the *third* line of output, j varies between 1 and 3. This continues until i takes on its final value of 6 so that j varies between 1 and 7.

The body of the inner for-loop, in line 3, consists of a print() statement that prints the value of j and suppresses the newline character (i.e., the optional argument end is set to the empty string). Following the inner loop, in line 4, is a print() statement with no arguments. This is used simply to generate the newline character. This print() statement is outside the body of the inner loop but inside the body of the outer sloop. Thus, this statement is executed seven times: once for each line of output.[1]

Changing gears a bit, consider the following collection of characters. This again consists of seven lines of output. The first line has a single character and each successive line has one additional character.

```
&
&&
&&&
&&&&
&&&&&
&&&&&&
&&&&&&&
```

How do you implement this? You can use nested loops, but Python actually provides a way to generate this using a single for-loop. To do so, you need to recall string repetition which was introduced in Sec. 5.6. When a string is "multiplied" by an integer, a new string is produced that is the original string repeated the number of times given by the integer. So, for example, "q" * 3 evaluates to the string "qqq". Listing 7.2 shows two implementations that generate the collection of ampersands shown above: one implementation uses a single loop while the other uses nested loops.

Listing 7.2 A triangle of ampersands generated using a single for-loop or nested for-loops. The implementation with a single loop takes advantage of Python's string repetition capabilities.

```
>>> for i in range(7):         # Seven lines of output.
...     print("&" * (i + 1)) # Num. characters increases as i increases.
...
&
&&
&&&
&&&&
&&&&&
&&&&&&
&&&&&&&
>>> for i in range(7):           # Seven lines of output.
...     for j in range(i + 1): # Inner loop for ampersands.
...         print("&", end="")
...     print()                  # Newline.
...
&
&&
&&&
```

[1] It may be worth mentioning that it is not strictly necessary to use two for-loops to obtain the output shown in Listing 7.1. As the code in the coming discussion suggests (but doesn't fully describe), it is possible to obtain the same output using a single for-loop (but then one has to provide a bit more code to construct the string that should appear on each line).

```
19 &&&&
20 &&&&&
21 &&&&&&
22 &&&&&&&
```

What if we wanted to invert this triangle so that the first line is the longest (with seven characters) and the last line is the shortest (with one character)? The code in Listing 7.3 provides a solution that uses a single loop. (Certainly other solutions are possible. For example, the `range()` function in the header of the `for`-loop can be used to directly generate the multipliers, i.e., integers that range from `7` to `1`. Then, the resulting loop variable can directly “multiply” the ampersand in the `print()` statement.)

Listing 7.3 An “inverted triangle” realized using a single `for`-loop.

```
1  >>> for i in range(7):        # Seven lines of output.
2  ...     print("&" * (7 - i)) # Num. characters decreases as i increases.
3  ...
4  &&&&&&&
5  &&&&&&
6  &&&&&
7  &&&&
8  &&&
9  &&
10 &
```

The header in line 1 is the same as the ones used previously: the loop variable `i` still varies between `0` and `6`. In line 2 the number of repetitions of the ampersand is `7 - i`. Thus, as `i` increases, the number of ampersands decreases.

As another example, consider the code given in Listing 7.4. The body of the outer `for`-loop contains, in line 2, a `print()` statement similar to the one in Listing 7.3 that was used to generate the inverted triangle of ampersands. Here, however, the newline character at the end of the line is suppressed. Next, in lines 3 and 4, a `for`-loop renders integers as was done in Listing 7.1. Outside the body of this inner loop a `print()` statement (line 5) simply generates a new line. Combining the inverted triangle of ampersands with the upright triangle of integers results in the rectangular collection of characters shown in lines 7 through 13.

Listing 7.4 An inverted triangle of ampersands is combined with an upright triangle of integers to form a rectangular structure of characters.

```
1  >>> for i in range(7):                  # Seven lines of output.
2  ...     print("&" * (7 - i), end="")  # Generate ampersands.
3  ...     for j in range(1, i + 2):       # Inner loop to display digits.
4  ...         print(j, end="")
5  ...     print()                         # Newline.
6  ...
```

```
 7  &&&&&&&1
 8  &&&&&&12
 9  &&&&&123
10  &&&&1234
11  &&&12345
12  &&123456
13  &1234567
```

Using similar code, let's construct an upright *pyramid* consisting solely of integers (and blank spaces). This can be realized with the code in Listing 7.5. The first four lines of Listing 7.5 are identical to those of Listing 7.4 except the ampersand in line 2 has been replaced by a blank space. In Listing 7.5, the `for`-loop that generates integers of increasing value (i.e., the loop in lines 3 and 4), is followed by the `for`-loop that generates integers of decreasing value (lines 5 and 6). In a sense, the values generated by this second loop are tacked onto the right side of the rectangular figure that was generated in Listing 7.4.

Listing 7.5 Pyramid of integers that is constructed with an outer `for`-loop and two inner `for`-loops. The first inner loop, starting on line 3, generates integers of increasing value while the second loop, starting on line 5, generates integers of decreasing value.

```
 1  >>> for i in range(7):
 2  ...     print(" " * (7 - i), end="") # Generate leading spaces.
 3  ...     for j in range(1, i + 2):     # Generate 1 through peak value.
 4  ...         print(j, end="")
 5  ...     for j in range(i, 0, -1):     # Generate peak - 1 through 1.
 6  ...         print(j, end="")
 7  ...     print()                       # Newline.
 8  ...
 9         1
10        121
11       12321
12      1234321
13     123454321
14    12345654321
15   1234567654321
```

Let's consider one more example of nested loops. Here, unlike in the previous examples, the loop variable for the outer loop does *not* appear in the header of the inner loop. Let's write a function that shows, as an ordered pair, the row and column numbers of positions in a table or matrix. Let's call this function `matrix_indices()`. It has two parameters corresponding to the number of rows and number of columns, respectively. In any previous experience you may have had with tables or matrices, the row and column numbers almost certainly started with one. Here, however, we use the numbering convention that is used for `lists`: the first row and column have an index of zero.

Listing 7.6 gives a function that generates the desired output.[2]

Listing 7.6 Function to display the row and column indices for a two-dimensional table or matrix where the row and column numbers start at zero.

```
1  >>> def matrix_indices(nrow, ncol):
2  ...     for i in range(nrow):      # Loop over the rows.
3  ...         for j in range(ncol): # Loop over the columns.
4  ...             print("(", i, ", ", j, ")", sep="", end=" ")
5  ...         print()
6  ...
7  >>> matrix_indices(3, 5)   # Three rows and five columns.
8  (0, 0) (0, 1) (0, 2) (0, 3) (0, 4)
9  (1, 0) (1, 1) (1, 2) (1, 3) (1, 4)
10 (2, 0) (2, 1) (2, 2) (2, 3) (2, 4)
11 >>> matrix_indices(5, 3)   # Five rows and three columns.
12 (0, 0) (0, 1) (0, 2)
13 (1, 0) (1, 1) (1, 2)
14 (2, 0) (2, 1) (2, 2)
15 (3, 0) (3, 1) (3, 2)
16 (4, 0) (4, 1) (4, 2)
```

The function defined in lines 1 through 5 has two parameters that are named nrow and ncol, corresponding to the desired number of rows and columns, respectively. nrow is used in the header of the outer loop in line 2 and ncol is used in the header of the inner loop in line 3.

In line 7 matrix_indices() is called to generate the ordered pairs for a matrix with three rows and five columns. The output appears in lines 8 through 10. In line 11 the function is called to generate the ordered pairs for a matrix with five rows and three columns.

In all the examples in this section the headers of the for-loops have used range() to set the loop variable to appropriate integer values. There is, however, another way in which nested for-loops can be constructed so that the iterables appearing in the headers are lists. This is considered in the next section.

7.2 lists of lists

A list can contain a list as an element or, in fact, contain any number of lists as elements. When one list is contained within another, we refer to this as *nesting* or we may say that one list is *embedded* within another. We may also refer to an *inner* list which is contained in a surrounding *outer* list. Nesting can be done to any level. Thus, for example, you can have a list that contains a list that contains a list and so on.

Listing 7.7 illustrates the nesting of one list within another.

[2] Unfortunately, if the user specifies that the number of rows or columns is greater than 11 (so that the row or column indices have more than one digit), the ordered pairs will no longer line up as nicely as shown here. When we cover string formatting, we will see ways to ensure the output is formatted “nicely” even for multiple digits.

Listing 7.7 Demonstration of nesting of one `list` as an element of another.

```
1  >>> # Create list with a string and a nested list of two strings.
2  >>> al = ['Weird Al', ['Like a Surgeon', 'Perform this Way']]
3  >>> len(al)   # Check length of al.
4  2
5  >>> al[0]
6  'Weird Al'
7  >>> al[1]
8  ['Like a Surgeon', 'Perform this Way']
9  >>> for item in al:   # Cycle through the elements of list al.
10 ...     print(item)
11 ...
12 Weird Al
13 ['Like a Surgeon', 'Perform this Way']
```

In line 2 the `list al` is defined with two elements. The first element is the string `'Weird Al'` and the second is a `list` that has two elements, both of which are themselves strings. The creation of this `list` immediately raises a question: How many elements does it have? Reasonable arguments can be made for either two or three but, in fact, as shown in lines 3 and 4, the `len()` function reports that there are two elements in `al`. Lines 5 and 6 show the first element of `al` and lines 7 and 8 show the second element, i.e., the second element of `al` is itself a complete two-element `list`. As before, a `for`-loop can be used to cycle through the elements of a `list`. This is illustrated in lines 9 through 13 where the `list al` is given as the iterable in the header.

Listing 7.8 provides another example of nesting one `list` within another; however, here there are actually three `lists` contained within the surrounding outer `list`. Importantly, as seen in lines 3 through 5, this code also demonstrates that the contents of a `list` can span multiple lines. The open bracket (`[`) acts similarly to open parentheses—it tells Python there is more to come. Thus, the `list` can be closed (with the closing bracket) on a subsequent line.[3]

Listing 7.8 Nesting of multiple `lists` inside a `list`. Here the `list produce` consists of `lists` that each contain a string as well as either one or two integers. The contents of a `list` may span multiple lines since an open bracket serves as a multi-line delimiter in the same way as an open parenthesis.

```
1  >>> # Create a list of three nested lists, each of which contains
2  >>> # a string and one or two integers.
3  >>> produce = [['carrots', 56],
4  ...            ['celery', 178, 198],
5  ...            ['bananas', 59]]
```

[3] Note, however, that any line breaks in the `list` must be between elements. One cannot, for instance, have a string that spans multiple lines just because it is enclosed in brackets. A string that spans multiple lines may appear in a `list` but it must adhere to the rules governing a multi-line string, i.e., it must be enclosed in triple quotes or the newline character at the end of each line must be escaped.

```
 6 >>> print(produce)
 7 [[’carrots’, 56], [’celery’, 178, 198], [’bananas’, 59]]
 8 >>> for i in range(len(produce)):
 9 ...     print(i, produce[i])
10 ...
11 0 [’carrots’, 56]
12 1 [’celery’, 178, 198]
13 2 [’bananas’, 59]
```

In line 3 the list produce is created. Each element of this list is itself a list. These inner lists are composed of a string and one or two integers. The string corresponds to a type of produce and the integer might represent the price per pound (in cents) of this produce. When there is more than one integer, this might represent the price at different stores. The print() statement in line 6 displays the entire list as shown in line 7. The for-loop in lines 8 and 9 uses indexing to show the elements of the outer list together with the index of the element.

Let’s consider a slightly more complicated example in which we create a list that contains three lists, each of which contains another list! The code is shown in Listing 7.9 and is discussed following the listing.

Listing 7.9 Creation of a list within a list within a list.

```
 1 >>> # Create individual artists as lists consisting of a name and a
 2 >>> # list of songs.
 3 >>> al = ["Weird Al", ["Like a Surgeon", "Perform this Way"]]
 4 >>> gaga = ["Lady Gaga", ["Bad Romance", "Born this Way"]]
 5 >>> madonna = ["Madonna", ["Like a Virgin", "Papa Don’t Preach"]]
 6 >>> # Collect individual artists together in one list of artists.
 7 >>> artists = [al, gaga, madonna]
 8 >>> print(artists)
 9 [[’Weird Al’, [’Like a Surgeon’, ’Perform this Way’]], [’Lady Gaga’,
10  [’Bad Romance’, ’Born this Way’]], [’Madonna’, [’Like a Virgin’,
11  "Papa Don’t Preach"]]]
12 >>> for i in range(len(artists)):
13 ...     print(i, artists[i])
14 ...
15 0 [’Weird Al’, [’Like a Surgeon’, ’Perform this Way’]]
16 1 [’Lady Gaga’, [’Bad Romance’, ’Born this Way’]]
17 2 [’Madonna’, [’Like a Virgin’, "Papa Don’t Preach"]]
```

In lines 3 through 5, the lists al, gaga, and madonna are created. Each of these lists consists of a string (representing the name of an artist) and a list (where the list contains two strings that are the titles of songs by these artists). In line 7 the list artists is created. It consists of the three lists of individual artists. The print() statement in line 8 is used to print the artists lists.[4] The for-loop in lines 12 and 13 is used to display the list corresponding to each individual artist.

[4] Line breaks have been added to the output to aid readability. In the interactive environment the output would

7.2.1 Indexing Embedded lists

We know how to index the elements of a list, but now the question is: How do you index the elements of a list that is embedded within another list? To do this you simply add another set of brackets and specify within these brackets the index of the desired element. So, for example, if the third element of xlist is itself a list, the second element of this embedded list is given by xlist[2][1]. The code in Listing 7.10 illustrates this type of indexing.

Listing 7.10 Demonstration of the use of multiple brackets to access an element of a nested list.

```
 1 >>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
 2 >>> toyota[0]
 3 ’Toyota’
 4 >>> toyota[1]
 5 [’Prius’, ’4Runner’, ’Sienna’, ’Camry’]
 6 >>> toyota[1][0]
 7 ’Prius’
 8 >>> toyota[1][3]
 9 ’Camry’
10 >>> len(toyota)      # What is length of outer list?
11 2
12 >>> len(toyota[1])   # What is length of embedded list?
13 4
14 >>> toyota[1][len(toyota[1]) - 1]
15 ’Camry’
```

In line 1 the list toyota is created with two elements: a string and an embedded list of four strings. Lines 2 and 3 display the first element of toyota. In lines 4 and 5 we see that the second element of toyota is itself a list. In line 6 two sets of brackets are used to specify the desired element. Interpreting these from right to left, these brackets (and the integers they enclose) specify that we want the first element of the second element of toyota. The first set of brackets (i.e., the left-most brackets) contains the index 1, indicating the second element of toyota, while the second set of brackets contains the index 0, indicating the first element of the embedded list. Despite the fact that we (humans) might read or interpret brackets from right to left, Python evaluates multiple brackets from left to right. So, in a sense, you can think of toyota[1][0] as being equivalent to (toyota[1])[0], i.e., first we obtain the list toyota[1] and then from this we obtain the first element. You can, in fact, add parentheses in this way, but it isn’t necessary or recommended. You should, instead, become familiar and comfortable with this form of multi-index notation as it is used in many computer languages.

Lines 10 and 11 of Listing 7.10 show the length of the toyota list is 2—as before, the embedded list counts as one element. Lines 12 and 13 show the length of the embedded list is 4. Line 14 uses a rather general approach to obtain the last element of a list (which here

be wrapped around at the border of the screen. This wrapping is not because of newline characters embedded in the output, but rather is a consequence of the way text is handled that is wider than can be displayed on a single line. Thus, if the screen size changes, the location of the wrapping changes correspondingly.

happens to be the embedded list given by toyota[1]). This approach, in which we subtract 1 from the length of the list, is really no different from the approach first demonstrated in Listing 6.7 for accessing the last element of any list.

As you may have guessed, a for-loop can be used to cycle through the elements of an embedded list. This is illustrated in the code in Listing 7.11.

Listing 7.11 Demonstration of cycling through the elements of an embedded list using a for-loop.

```
1  >>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
2  >>> for model in toyota[1]: # Cycle through embedded list.
3  ...     print(model)
4  ...
5  Prius
6  4Runner
7  Sienna
8  Camry
```

Line 1 again creates the toyota list with an embedded list as its second element. The for-loop in lines 2 and 3 prints each element of this embedded list (which corresponds to a Toyota car model).

Let us expand on this a bit and write a function that displays a car manufacturer (i.e., a brand or make) and the models made by each manufacturer (or at least a subset of the models—we won't bother trying to list them all). It is assumed that manufacturers are organized in lists where the first element is a string giving the brand name and the second element is a list of strings giving model names. Listing 7.12 provides the code for this function as well as examples of its use. The code is described following the listing.

Listing 7.12 A function to display the make and list of models of a car manufacturer.

```
>>> def show_brand(brand):
...     print("Make:", brand[0])
...     print("  Model:")
...     for i in range(len(brand[1])):
...         print("    ", i + 1, brand[1][i])
...
>>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
>>> ford = ["Ford", ["Focus", "Taurus", "Mustang", "Fusion", "Fiesta"]]
>>> show_brand(toyota)
Make: Toyota
  Model:
     1 Prius
     2 4Runner
     3 Sienna
     4 Camry
>>> show_brand(ford)
```

```
17 Make: Ford
18   Model:
19      1 Focus
20      2 Taurus
21      3 Mustang
22      4 Fusion
23      5 Fiesta
```

The show_brand() function is defined in lines 1 through 5. This function takes a single argument, named brand, that is a list that contains the make and models for a particular brand of car. Line 2 prints the make of the car while line 3 prints a label announcing that what follows are the various models. Then, in lines 4 and 5, a for-loop cycles through the second element of the brand list, i.e., cycles through the models. The print() statement in line 5 has four blank spaces, then a counter (corresponding to the element index plus one), and then the model. Lines 7 and 8 define lists that are appropriate for the Totoya and Ford brands of cars. The remaining lines show the output produced when these lists are passed to the show_brand() function.

Listing 7.13 presents the function show_nested_lists() that can be used to display all the (inner) elements of a list of lists. This function has a single parameter which is assumed to be the outer list.

Listing 7.13 Function to display the elements of lists that are nested within another list.

```
 1 >>> def show_nested_lists(xlist):
 2 ...     for i in range(len(xlist)):
 3 ...         for j in range(len(xlist[i])):
 4 ...             print("xlist[", i, "][", j, "] = ", xlist[i][j], sep="")
 5 ...         print()
 6 ...
 7 >>> produce = [['carrots', 56], ['celery', 178, 198], ['bananas', 59]]
 8 >>> show_nested_lists(produce)
 9 xlist[0][0] = carrots
10 xlist[0][1] = 56
11
12 xlist[1][0] = celery
13 xlist[1][1] = 178
14 xlist[1][2] = 198
15
16 xlist[2][0] = bananas
17 xlist[2][1] = 59
```

The function is defined in lines 1 through 5. The sole parameter is named xlist. The for-loop whose header is in line 2 cycles through the indices for xlist. On the other hand, the for-loop with the header in line 3 cycles through the indices that are valid for the lists nested within xlist. The body of this inner for-loop consists of a single print() statement that incorporates the indices as well as the contents of the element. Importantly, note that the nested lists do not have to be the same length. The first and third elements of xlist are each two-element lists. However, the second element of xlist (corresponding to the list with "celery") has three elements.

7.2.2 Simultaneous Assignment and lists of lists

We have seen that lists can be used with simultaneous assignment (Sec. 6.8). When dealing with embedded lists, simultaneous assignment is sometimes used to write code that is much more readable than code that does not employ simultaneous assignment. For example, in the code in Listings 7.11 and 7.12 the list toyota was created. The element toyota[0] contains the make while toyota[1] contains the models. When writing a program that may have multiple functions and many lines of code, it may be easy to lose sight of the fact that a particular element maps to a particular thing. One way to help alleviate this is to “unpack” a list into parts that are associated with appropriately named variables. This unpacking can be done with simultaneous assignment. Listing 7.14 provides an example.[5]

Listing 7.14 Demonstration of “unpacking” a list that contains an embedded list. Simultaneous assignment is used to assign the elements of the toyota list to appropriately named variables.

```
1  >>> toyota = ["Toyota", ["Prius", "4Runner", "Sienna", "Camry"]]
2  >>> make, models = toyota     # Simultaneous assignment.
3  >>> make
4  'Toyota'
5  >>> for model in models:
6  ...     print("    ", model)
7  ...
8       Prius
9       4Runner
10      Sienna
11      Camry
```

In line 1 the toyota list is created with the brand name (i.e., the make) and an embedded list of models. Line 2 uses simultaneous assignment to “unpack” this two-element list to two appropriately named variables, i.e., make and models. Lines 3 and 4 show make is correctly set. The for-loop in lines 5 and 6 cycles through the models list to produce the output shown in lines 8 through 11.

Let us consider another example where the goal now is to write a function that cycles through a list of artists similar to the list that was constructed in Listing 7.9. Let’s call the function show_artists(). This function should cycle through each element (i.e., each artist) of the list it is passed. For each artist it should display the name and the songs associated with the artist. Listing 7.15 shows a suitable implementation of this function.

[5] This is somewhat contrived in that a list is created and then immediately unpacked. Try instead to imagine this type of unpacking being done in the context of a much larger program with multiple lists of brands and lists being passed to functions.

Listing 7.15 A function to display a list of artists in which each element of the artists list contains an artist's name and a list of songs.

```
1  >>> def show_artists(artists):
2  ...     for artist in artists:       # Loop over the artists.
3  ...         name, songs = artist     # Unpack name and songs.
4  ...         print(name)
5  ...         for song in songs:       # Loop over the songs.
6  ...             print("    ", song)
7  ...
8  >>> # Create a list of artists.
9  >>> performers = [
10 ...     ["Weird Al", ["Like a Surgeon", "Perform this Way"]],
11 ...     ["Lady Gaga", ["Bad Romance", "Born this Way"]],
12 ...     ["Madonna", ["Like a Virgin", "Papa Don't Preach"]]]
13 >>> show_artists(performers)
14 Weird Al
15      Like a Surgeon
16      Perform This Way
17 Lady Gaga
18      Bad Romance
19      Born This Way
20 Madonna
21      Like a Virgin
22      Papa Don't Preach
```

The show_artists() function is defined in lines 1 through 6. Its sole parameter is named artists (plural). This parameter is subsequently used as the iterable in the header of the for-loop in line 2. The loop variable for this outer loop is artist (singular). Thus, given the presumed structure of the artists list, for each pass of the outer loop, the loop variable artist corresponds to a two-element list containing the artist's name and a list of songs. In line 3 the loop variable artist is unpacked into a name and a list of songs. Note that we don't explicitly know from the code itself that, for example, songs is a list nor even that artist is a two-element list. Rather, this code relies on the presumed structure of the list that is passed as an argument to show_artists(). Having obtained the name and songs, these values are displayed using the print() statement in line 4 and the for-loop in lines 5 and 6.

Following the definition of the function, in lines 9 through 12, a list named performers is created that is suitable for passing to show_artists(). The remainder of the listing shows the output produced when show_artists() is passed the performers list.

Perhaps somewhat surprisingly, simultaneous assignment can be used directly in the header of a for-loop. To accomplish this, each item of the iterable in the header must have nested within it as many elements as there are lvalues to the left of the keyword in. Listing 7.16 provides a template for using simultaneous assignment in a for-loop header. In the header, there are N comma-separated lvalues to the left of the keyword in. There must also be N elements nested within each element of the iterable.

Listing 7.16 Template for a `for`-loop that employs simultaneous assignment in the header.

```
for <item1>, ..., <itemN> in <iterable>:
    <body>
```

Listing 7.17 provides a demonstration of this form of simultaneous assignment. In lines 1 through 3 a `list` is created of shoe designers and, supposedly, the cost of a pair of their shoes. Note that each element of the `shoes list` is itself a two-element `list`. The discussion of the code continues following the listing.

Listing 7.17 Demonstration of simultaneous assignment used in the header of a `for`-loop. This sets the value of two loop variables in accordance with the contents of the two-element `lists` that are nested in the iterable (i.e., nested in the `list shoes`).

```
1  >>> shoes = [["Manolo Blahnik", 120], ["Bontoni", 96],
2  ...     ["Maud Frizon", 210], ["Tinker Hatfield", 54],
3  ...     ["Lauren Jones", 88], ["Beatrix Ong", 150]]
4  >>> for designer, price in shoes:
5  ...     print(designer, ": $", price, sep="")
6  ...
7  Manolo Blahnik: $120
8  Bontoni: $96
9  Maud Frizon: $210
10 Tinker Hatfield: $54
11 Lauren Jones: $88
12 Beatrix Ong: $150
```

The `header` of the `for`-loop in line 4 uses simultaneous assignment to assign a value to both the loop variables `designer` and `price`. So, for example, prior to the first pass through the body of the loop it is as if this statement had been issued:

```
    designer, price = shoes[0]
```

or, thought of in another way

```
    designer, price = ["Manolo Blahnik", 120]
```

Then, prior to the second pass through the body of the `for`-loop, it is as if this statement had been issued

```
    designer, price =  ["Bontoni", 96]
```

And so on. The body of the `for`-loop consists of a `print()` statement that displays the designer and the associated price (complete with a dollar sign).

As illustrated in Listing 7.18, this form of simultaneous assignment works when dealing with `tuples` as well. In line 2 a `list` of two-element `tuples` is created. The `for`-loop in lines 3 and 4 cycles through these tuples, assigning the first element of the tuple to count and the second element of the tuple to fruit. The body of the loop prints these values.

Listing 7.18 Demonstration that tuples and lists behave in the same way when it comes to nesting and simultaneous assignment in for-loop headers.

```
 1 >>> # Create a list of tuples.
 2 >>> flist = [(21, ’apples’), (17, ’bananas’), (39, ’oranges’)]
 3 >>> for count, fruit in flist:
 4 ...     print(count, fruit)
 5 ...
 6 21 apples
 7 17 bananas
 8 39 oranges
 9 >>> # Create a tuple of tuples.
10 >>> ftuple = ((21, ’apples’), (17, ’bananas’), (39, ’oranges’))
11 >>> for count, fruit in ftuple:
12 ...     print(count, fruit)
13 ...
14 21 apples
15 17 bananas
16 39 oranges
```

Lines 10 through 16 are nearly identical to lines 2 through 8. The only difference is that the data are collected in a tuple of tuples rather than a list of tuples.

7.3 References and list Mutability

In Sec. 4.4 we discussed the scope of variables. We mentioned that we typically want to pass data into a function via the parameters and obtain data from a function using a return statement. However, because a list is a mutable object, the way it behaves when passed to a function is somewhat different than what we have seen before. In fact, we don’t even have to pass a list to a function to observe this different behavior. In this section we want to explore this behavior. The primary goal of the material presented here is to help you understand the cause of bugs that may appear in your code. We do not seek to fully flesh out the intricacies of data manipulation and storage in Python.

First, recall that when we assign a value, such as an integer or float, to a variable and then assign that variable to a second variable, changes to the first variable do not subsequently affect the second variable (or vice versa). This is illustrated in the following (please read the comments embedded in the code):

```
>>> x = 11         # Assign 11 to x.
>>> y = x          # Assign value of x to y
>>> print(x, y)    # Show x and y.
11 11
```

```
>>> x = 22          # Change x to 22.
>>> print(x, y)     # Show that x has changed, but not y.
22 11
>>> y = 33          # Now change y to 33.
>>> print(x, y)     # Show that y has changed, but not x.
22 33
```

This is *not* how `lists` behave. In Python a `list` may have hundreds, thousands, or even millions of elements—the data within a `list` may occupy a significant amount of computer memory. Thus, when a `list` is assigned to a new variable the default action is *not* to make an independent copy of the underlying data for this new variable. Instead, the assignment makes the new variable a *reference* (or an *alias*) that points to the same underlying data (i.e., it points to the same memory location where the original `list` is stored). This is demonstrated in Listing 7.19 which parallels the code above except now the data is contained in a `list`.

Listing 7.19 Demonstration that when a `list` is assigned to a variable, the variable serves as a reference (or alias) to the underlying data in memory.

```
1  >>> xlist = [11]          # Create xlist with a single element.
2  >>> ylist = xlist         # Make ylist an alias for xlist.
3  >>> print(xlist, ylist)   # Show xlist and ylist.
4  [11] [11]
5  >>> xlist[0] = 22         # Change value of the element in xlist.
6  >>> print(xlist, ylist)   # Change is visible in both xlist and ylist.
7  [22] [22]
8  >>> ylist[0] = 33         # Change value of the element in ylist.
9  >>> print(xlist, ylist)   # Change is visible in both xlist and ylist.
10 [33] [33]
11 >>> # Use built-in is operator to show xlist and ylist are the same.
12 >>> xlist is ylist
13 True
```

In line 1 a single-element `list` is created and assigned to the variable `xlist`. In line 2 `xlist` is assigned to `ylist`. This assignment makes `ylist` point to the same location in the computer's memory that `xlist` points to, i.e., they both point to the single-element `list` that contains the integer `11`. The `print()` statement in line 3 and the subsequent output in line 4 show that the contents of `xlist` and `ylist` are the same. In line 5 the value of the element in `xlist` is changed to `22`. The `print()` statement in line 6 shows this change is reflected in the contents of both `xlist` and `ylist`! In line 8 the value of the element within `ylist` is changed to `33`. The subsequent `print()` statement in line 9 shows this change is also evident in both `lists`. This behavior is a consequence of the fact that there is really only one location in memory where this `list` is stored and both `xlist` and `ylist` point to this location.

There is a built-in operator called `is` that can be used to determine whether two variables point to the same memory location. The `is` operator is used in line 12 to ask if `xlist` and `ylist` point to the same memory location. The answer, shown on line 13, is `True`—they do point to the same memory location. Thus, because *a list is mutable*, a change to the list made using one of the variables will affect the underlying data seen by both variables.

Now, let's consider code that may initially look the same as the code in Listing 7.19; however, there is an important distinction. The first four lines of Listing 7.20 are identical to those of Listing 7.19. A single-element list is created and assigned to the variable xlist which, in turn, is assigned to ylist in line 2. The is operator is used in line 5 to show that xlist and ylist are aliases for the same data. The discussion continues following the listing.

Listing 7.20 As demonstrated in the following, although two variables may initially be aliases for the same underlying data, if one of the variables is assigned new data, this will not affect the original data.

```
1  >>> xlist = [11]          # Create xlist with a single element.
2  >>> ylist = xlist         # Make ylist an alias for xlist.
3  >>> print(xlist, ylist)  # Show xlist and ylist.
4  [11] [11]
5  >>> xlist is ylist        # See that xlist and ylist are aliases.
6  True
7  >>> xlist = [22]          # Change xlist to point to a new list.
8  >>> print(xlist, ylist)  # See that xlist changes, but not ylist.
9  [22] [11]
10 >>> xlist is ylist   # See that xlist and ylist are no longer the same.
11 False
```

In line 7 xlist is assigned to a new one-element list, i.e., xlist now points to a new list. Note, importantly, that in line 7 we are *not* changing the value of an element in the list that was assigned to xlist in line 1. (Please compare line 7 of Listing 7.20 to line 5 of Listing 7.19.) Instead, we are assigning xlist to an entirely new list. This assignment does not affect where ylist points in memory. Thus, the output of the print() statement in line 8 shows that xlist and ylist are now different. The is operator is used in line 10 to test if xlist and ylist are aliases for the same underlying data. The result of False in line 11 shows they are not.

With the understanding that assignment of a list to a variable actually creates a reference (or alias) to the underlying data in the computer's memory, let us consider passing a list to a function and the consequences of changing the elements of the list within the function. Listing 7.21 starts by defining a function in lines 1 through 3. This function takes a single argument which is assumed to be a list. The function contains a for-loop that multiplies each of the elements by 2 and assigns this value back to the original location in the list. Note that this function has *no* return statement. Thus, it is a void function—it does not return anything useful. Instead, we would call this function for its side effects, i.e., its ability to double the elements of a list. The discussion continues below the listing.

Listing 7.21 Demonstration that changes to a list that occur inside a function are visible outside the function.

```
1  >>> def doubler(xlist):
```

```
2  ...     for i in range(len(xlist)):
3  ...         xlist[i] = 2 * xlist[i]
4  ...
5  >>> zlist = [1, 2, 3]  # Create list of integers.
6  >>> print(zlist)       # Check contents.
7  [1, 2, 3]
8  >>> doubler(zlist)     # Call doubler().
9  >>> print(zlist)       # See that contents have doubled.
10 [2, 4, 6]
```

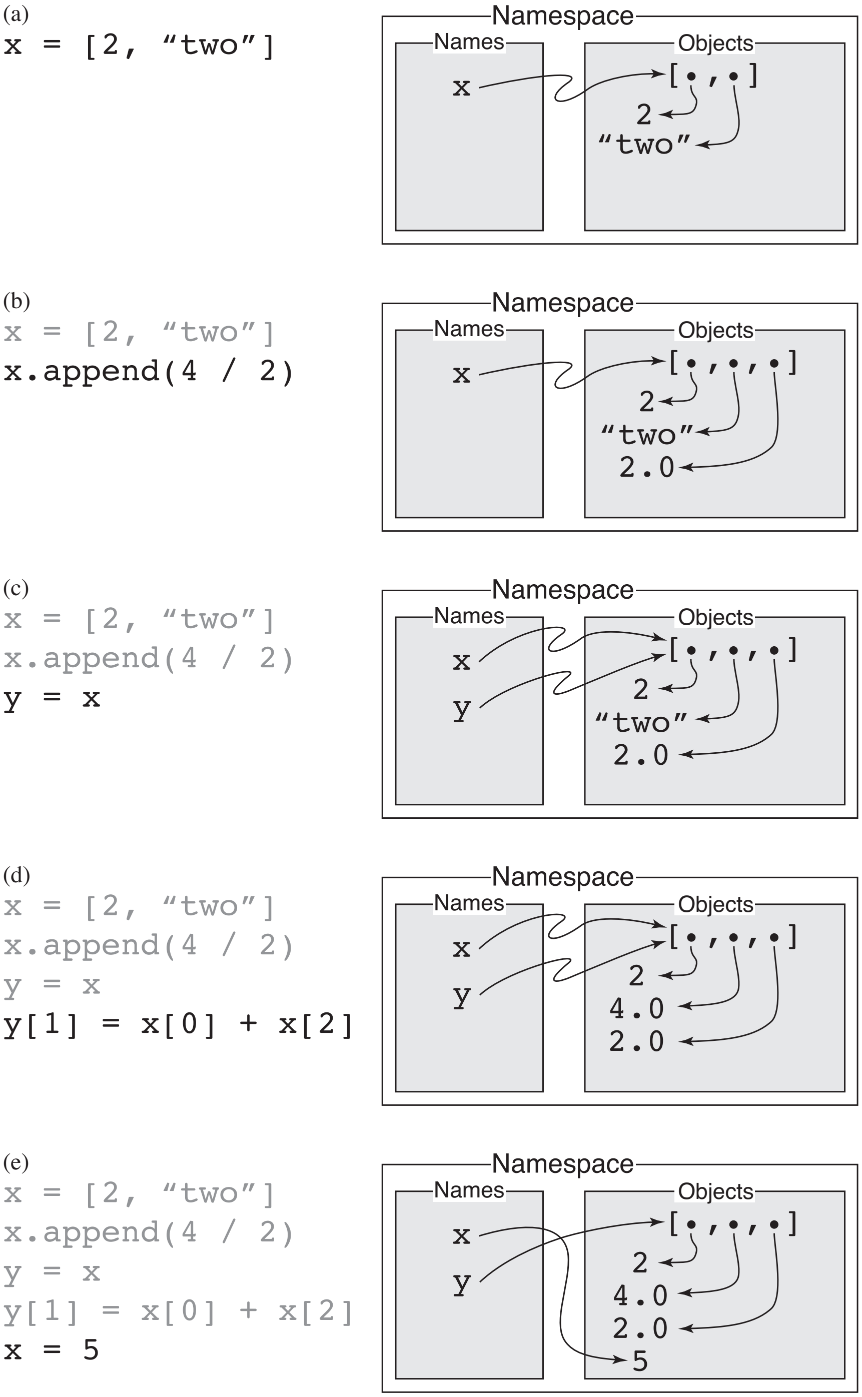
In line 5 `zlist` is created with three integer elements. The `print()` statement in line 6 displays the contents of the `list`. The `doubler()` function is called in line 8. Thus, within the function itself, `xlist` (the formal parameter of the function) serves as an alias for `zlist`. We see, with the `print()` statement in line 9, that the elements of `zlist` were indeed doubled.

To further illustrate how `lists` behave when involved in assignment operations, Fig. 7.1 depicts the contents of a namespace as various statements involving a `list` are executed. Recall that the "Name" portion of a namespace is a collection of identifiers and it is the role of the namespace to associate these identifiers with objects. In Python, a `list` is a form of "container." As such, each element of a `list` can reference (i.e., be associated with) any other type of object. In Fig. 7.1(a) we assume the statement `x = [2, "two"]` has been issued. The literal `list` `[2, "two"]` is stored in memory by creating a container of two elements. This is depicted by `[·,·]` where the dots can be references to any other object. The first element of this container refers to the integer `2` while the second element refers to the string `two`. The assignment operator is used to associate the indentifier `x` with this `list`.

Figure 7.1(b) indicates the effect of using the `append()` method to append an object to the `list`: the container grows by an additional element. Note that the expression that appears in the argument of the `append()` method is evaluated prior to appending the object to the end of the `list`. In Fig. 7.1(c) the identifier `y` is assigned `x`. This merely serves to associate `y` with the same object to which `x` currently refers. In Fig. 7.1(d) the second element of `y` is assigned `x` the sum of the first and third elements of `x`. But, there is only one `list` stored in memory so we would have used any combination of the identifiers `x` and `y` in this statement and the outcome would have been the same. In Fig. 7.1(e), the identifier `x` is assigned the integer value `5`. Thus, `x` no longer refers to the `list`, but this assignment does *not* affect the value of `y`.

Now consider the namespace depicted in Fig. 7.2 where the statement that gave rise to this arrangement of objects is given above the namespace. To the right of the assignment operator is a `list` of `lists`. Thus, some of the elements of the various containers refer to other containers. The object associated with any element of any of these `lists` can be changed because `lists` are mutable and there are no restrictions to what an element of a container refers. Furthermore, the lengths of the `lists` may be changed using the appropriate methods.[6]

[6] The methods `append()` and `extend()` will increase the length of a `list` while `pop()` and `remove()` can be used to remove elements from a `list` and hence decrease the length.



(a)

```
x = [2, “two”]
```

(b)

```
x = [2, “two”]
x.append(4 / 2)
```

(c)

```
x = [2, “two”]
x.append(4 / 2)
y = x
```

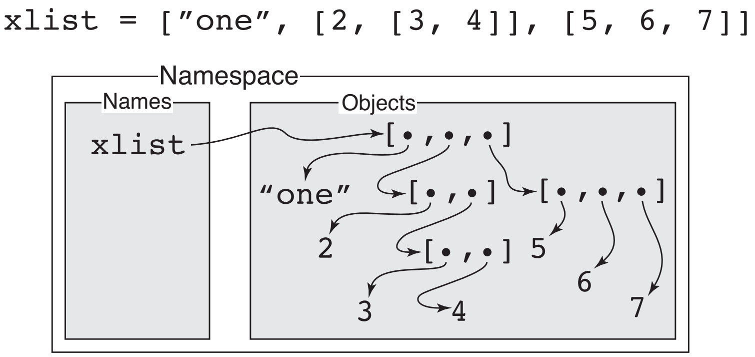
(d)

```
x = [2, “two”]
x.append(4 / 2)
y = x
y[1] = x[0] + x[2]
```

(e)

```
x = [2, “two”]
x.append(4 / 2)
y = x
y[1] = x[0] + x[2]
x = 5
```

Figure 7.1: Depiction of the behavior of a namespace when various statements are issued involving a `list`.



```
xlist = ["one", [2, [3, 4]], [5, 6, 7]]
```

Figure 7.2: Depiction of a namespace after the assignment of a `list` of `lists` to the identifier `xlist`. The statement corresponding to the depiction is shown above the namespace.

7.4 Strings as Iterables or Sequences

We will have much more to say about strings in Chap. 9, but it is appropriate now to consider a couple of aspects of strings. As you know, strings are a sequential collection of characters. As such, each character has a well defined position within the string. This is similar to the elements within `lists` and `tuples`. Thus, it may come as no surprise that we can access individual characters in a string by indicating the desired character with an integer index enclosed in square brackets following the string. As with `lists` and `tuples`, an index of `0` is used for the first character of a string—the index represents the offset from the beginning. Additionally, the `len()` function provides the length of a string, i.e., the total number of characters. The individual characters of a string are themselves strings (with lengths of 1).[7]

Listing 7.22 illustrates accessing individual characters of a string by explicit indexing.

Listing 7.22 Explicit indexing used to access the characters of a string. As with `tuples` and `lists`, the index represents the offset from the beginning.

```
>>> hi = "Hello World!"
>>> hi[0]                 # First character.
'H'
>>> hi[6]                 # Seventh character.
'W'
>>> len(hi)               # Length of string.
12
>>> hi[len(hi) - 1]       # Last character.
'!'
>>> print(hi[0], hi[1], hi[5], hi[7], hi[9], hi[10], hi[11], sep = "")
He old!
```

[7] In some computer languages individual characters have a different data type than strings, but this is *not* the case in Python.

Listing 7.23 demonstrates how a `for`-loop can be used with explicit indexing to loop over all the characters in a string. In line 1 the string `"Hey!"` is assigned to the variable `yo`. The header of the `for`-loop in line 2 is constructed to cycle through all the valid indices for the characters of `yo`. The body of the `for`-loop, in line 3, prints the index and the corresponding character.

Listing 7.23 Use of a `for`-loop and explicit indexing to loop over all the characters in a string.

```
1 >>> yo = "Hey!"
2 >>> for i in range(len(yo)):
3 ...     print(i, yo[i])
4 ...
5 0 H
6 1 e
7 2 y
8 3 !
```

There is another way to loop over the individual characters of a string. Another feature that strings share with `lists` and `tuples` is that they are iterables[8] and thus can be used directly as the iterable in the header of a `for`-loop. This is demonstrated in Listing 7.24.

Listing 7.24 Demonstration that a string can be used as the iterable in the header of a `for`-loop. When a string is used this way, the loop variable is successively set to each individual character of the string for each pass of the `for`-loop.

```
>>> yo = "Hey!"
>>> for ch in yo:
...     print(ch)
...
H
e
y
!
```

Note that individual characters of strings can be accessed via an index. Similarly, individual elements of `tuples` and `lists` can be accessed via an index. Objects whose contents can be accessed in this way are known as *sequences*.[9] Recall that a `tuple` differs from a `list` in that a `tuple` is immutable while a `list` is mutable (its elements can be changed). A string is similar to a `tuple` in that the individual characters of a string cannot be changed, i.e., a string is immutable.

[8] Technically, an iterable is any object that has either a `__iter__()` or `__getitem__()` method. Behind the scenes Python uses these methods to get the values of the iterable. But, this sort of detail really doesn't concern us.

[9] A sequence is an object that has a `__getitem__()` method. Thus, sequences are automatically iterables, but the converse is not true: not all iterables are sequences. Assume `s` is a sequence (whether a string, `list`, or `tuple`). When we write `s[i]` Python essentially translates this to `s.__getitem__(i)`. You can, in fact, use this method call yourself to confirm that the brackets are really just providing a shorthand for the calling of `__getitem__()`.

7.5 Negative Indices

In addition to the positive indexing we have been using, an element of a sequence can also be specified using a *negative* index. In terms of negative indexing, the last element of a sequence has an index of `-1`. When the last element of a sequence is desired, it is generally more convenient to use negative indexing. For example, if one is interested in the last element of the `list` `xlist`, it is certainly more convenient to write `xlist[-1]` than `xlist[len(xlist) - 1]`.

Listing 7.25 shows the positive and negative indices for the string `"Hello!"`. The positive indices vary between zero and one less than the length (corresponding to the first and last characters, respectively). The negative indices vary between the negative of the length to `-1` (again, corresponding to the first and last characters, respectively).

Listing 7.25 Positive and negative indices for the string `"Hello!"`.

```
    0    1    2    3    4    5    <=> Positive indices
  +---+---+---+---+---+---+
  | H | e | l | l | o | ! |
  +---+---+---+---+---+---+
   -6   -5   -4   -3   -2   -1    <=> Negative indices
```

Listing 7.26 demonstrates negative indexing to access characters of a string. A six-character string is created in line 1 and assigned to the variable `hi`. The first element is accessed in line 2 by explicitly putting the negative index corresponding to the first character. In line 4 the `len()` function is used to obtain an expression that, in general, produces the first character of a string of arbitrary length. In line 6 the last character of the string is accessed.

Listing 7.26 Demonstration of negative indexing for a string. Elements of a `tuple` or a `list` can also be accessed using negative indexing.

```
1  >>> hi = "Hello!"    # Create a six-character string.
2  >>> hi[-6]          # Explicitly access first element.
3  'H'
4  >>> hi[-len(hi)]    # General construct for accessing first element.
5  'H'
6  >>> hi[-1]          # Last element of string.
7  '!'
8  >>> for i in range(1, len(hi) + 1):
9  ...     print(-i, hi[-i])
10 ...
11 -1 !
12 -2 o
13 -3 l
14 -4 l
15 -5 e
16 -6 H
```

The for-loop in lines 8 and 9 cycles through all the negative indices which are displayed in the output together with the corresponding character.

Although these examples used strings, negative indexing behaves the same way when accessing the elements of tuples and lists.

7.6 Slicing

Slicing provides a convenient way to extract or access a subset of a sequence. In many ways slicing is akin to the two-argument or three-argument forms of the range() function. To obtain a slice of a sequence, one specifies an integer start value and an integer stop value. These values are given in square brackets following the sequence (where one would normally provide a single index). These start and stop values are separated by a colon. The resulting slice is the portion of the original sequence that starts from the index start and ends just before stop. For example, assuming xlist is a list and that the start and stop values are both non-negative, the expression

```
xlist[start : stop]
```

returns a new list with elements

```
[xlist[start], xlist[start + 1], ..., xlist[stop - 1]]
```

If start is omitted, it defaults to 0. If stop is omitted, it defaults to the length of the sequence.

As a slight twist on the description above, positive or negative indices can be used for either the start or stop value. As before, when one index is positive and the other negative, the resulting slice starts with the specified start index and progresses up to, but does not include, the stop index. However, when using indices of mixed sign, one cannot interpret the resulting slice in terms of the indexing shown above (where the index of successive elements is incremented by +1 from the start). Instead, one has to translate a negative index into its equivalent positive value. This fact will become more clear after considering a few examples.

Listing 7.27 provides various examples of slicing performed on a string. The comments in the code provide additional information.

Listing 7.27 Demonstration of slicing a string.

```
>>> #    0123456789012345  -> index indicator
>>> s = "Washington State" # Create a string.
>>> s[1 : 4]    # Second through fourth characters.
'ash'
>>> s[ : 4]     # Start through fourth character.
'Wash'
>>> s[4 : 6]    # Fifth and sixth characters.
'in'
>>> s[4 : ]     # Fifth character through the end.
'ington State'
>>> s[-3 : ]    # Third character from end to end.
```

```
12 'ate'
13 >>> # Fifth from start to (but not including) third from end.
14 >>> s[4 : -3]    # Equivalent to s[4 : 13]
15 'ington St'
16 >>> # Tenth from end through tenth from start.
17 >>> s[-10 : 10] # Equivalent to s[6 : 10]
18 'gton'
19 >>> s[-9 : -2]  # Negative start & stop.
20 'ton Sta'
21 >>> s[7 : 14]   # Positive start & stop; equiv. to previous expression.
22 'ton Sta'
23 >>> s[ : ]        # The entire sequence.
24 'Washington State'
```

In line 14 and line 17 slices are obtained that use indices with mixed signs. The slice of `s[4 : -3]` is equivalent to the slice `s[4 : 13]`. Note that if the `range()` function were passed these `start` and `stop` values, i.e., `range(4, -3)`, it would not produce any integers. Thus, although slicing is closely related to the `range()` function, there are some differences.

Note the expression in line 23. Both the `start` and `stop` values are omitted so that these take on the default values of `0` and the length of the string, respectively. Thus, the resulting slice, shown in line 24, is the entire string. This last expression may seem rather silly, but it can actually prove useful in that slices return copies of the original sequence. If the slice is specified by `[ : ]`, then Python will make a copy of the *entire* sequence. This can come in handy when one needs to make a copy of an entire `list`.

Listing 7.28 provides an example where an “original” `list` is assigned to `xlist` in line 1. A slice that spans all of `xlist` is assigned to `ylist` in line 2. In line 3 `zlist` is simply assigned the value `xlist`. As we know from Listing 7.19, `zlist` is an alias for `xlist`. But, `ylist` is a completely independent copy of the original data. As lines 6 through 11 illustrate, changes to `xlist` do not affect `ylist` and, likewise, changes to `ylist` do not affect `xlist`. The `is` operator is used in line 15 to confirm what we already know: `xlist` and `ylist` point to different memory locations.

Listing 7.28 Demonstration that a slice can be used to obtain an independent copy of an entire `list`.

```
1  >>> xlist = [1, 2, 3]    # Original list.
2  >>> ylist = xlist[ : ]   # Independent copy of list.
3  >>> zlist = xlist        # Alias for original list.
4  >>> print(xlist, ylist, zlist) # Show lists.
5  [1, 2, 3] [1, 2, 3] [1, 2, 3]
6  >>> xlist[1] = 200             # Change original list.
7  >>> print(xlist, ylist, zlist) # See change in original but not copy.
8  [1, 200, 3] [1, 2, 3] [1, 200, 3]
9  >>> ylist[2] = 300             # Change copy.
10 >>> print(xlist, ylist, zlist) # See change in copy but not "original."
11 [1, 200, 3] [1, 2, 300] [1, 200, 3]
```

```
>>> zlist[0] = 100            # Change original via the alias.
>>> print(xlist, ylist, zlist) # See change in original but not copy.
[100, 200, 3] [1, 2, 300] [100, 200, 3]
>>> xlist is ylist    # Use is to show original and copy are different.
False
```

A slice can be specified with three terms (or three arguments). The third term is the *step* or *increment* and it is separated from the `stop` value by a colon. So, for the `list xlist`, one can write

```
xlist[start : stop : increment]
```

When the increment is not provided, it defaults to `1`. The increment may be negative. When the increment is negative, this effectively inverts the default values for `start` and `stop`. For a negative increment the default `start` is `-1` (i.e., the last element or character) and the default `start` is the negative of the quantity length plus one (e.g., `-(len(xlist) + 1))`. As will be demonstrated in a moment, by doing this, if one uses an increment of `-1` and the default `start` and `stop`, the entire sequence is inverted.

Listing 7.29 demonstrates slicing where an increment is specified. In lines 2, 4, and 7 an increment of `2` is used to obtain every other character over the specified `start` and `stop` values. In lines 9, 11, and 13 the increment is `-1` so that characters are displayed in reverse order. The expression in line 9 inverts the entire string as does the expression in line 13. However the expression in line 9 relies on the default `start` and `stop` values while the expression in line 13 gives these explicitly.

Listing 7.29 Demonstration of slicing where the increment is provided as the third term in brackets.

```
1  >>> s = "Washington State"
2  >>> s[ : : 2]      # Every other character from the start.
3  'Wsigo tt'
4  >>> s[ 5 : : 2]    # Every other character from the sixth.
5  'ntnSae'
6  >>> # Every other character from sixth, excluding last character.
7  >>> s[ 5 : -1 : 2]
8  'ntnSa'
9  >>> s[ : : -1]     # Negative increment -- invert string.
10 'etatS notgnihsaW'
11 >>> s[10 : 1 : -1] # Eleventh character to third; negative increment.
12 ' notgnihs'
13 >>> s[-1 : -len(s) - 1 : -1]
14 'etatS notgnihsaW'
```

For slices it is not an error to specify `start` or `stop` values that are out of range, i.e., specify indices that are before the beginning or beyond the end of the given sequence. It is, however, an error to specify an individual index that is out of range. This is demonstrated in Listing 7.30. A three-element list is created in line 1. In line 2 a slice is created that specifies the start is the second element and the stop value is beyond the end of the list. The result, shown in line 2, starts at the second element and goes until the end of the list. Similarly, in line 4, the start value is before the first element and the stop value says to stop before the second element. The result, in line 5, is simply the first element of the list. In line 6 the start value is before the start of the list and the stop value is beyond the end. In this case, the result is the entire list as shown in line 7. The next two statements, in lines 8 and 12, show that if we try to access an individual element outside the range of valid indices, we obtain an IndexError.

Listing 7.30 Demonstration that out-of-range values can be given for a slice but cannot be given for an individual item in a sequence.

```
 1  >>> xlist = ['a', 'b', 'c']
 2  >>> xlist[1 : 100]          # Slice: Stop value beyond end.
 3  ['b', 'c']
 4  >>> xlist[-100 : 1]         # Slice: Start value before beginning.
 5  ['a']
 6  >>> xlist[-100 : 100]       # Slice: Start and stop out of range.
 7  ['a', 'b', 'c']
 8  >>> xlist[-100]             # Individual element out of range.
 9  Traceback (most recent call last):
10    File "<stdin>", line 1, in <module>
11  IndexError: list index out of range
12  >>> xlist[100]              # Individual element out of range.
13  Traceback (most recent call last):
14    File "<stdin>", line 1, in <module>
15  IndexError: list index out of range
```

7.7 list Comprehension (optional)

We often need to iterate through the elements of a list, perform operation(s) on those elements, and store the resulting values in a new list, leaving the original list unchanged. Although this can be accomplished using a for-loop, to make this pattern easier to implement and more concise, Python provides a construct known as *list comprehension*. The syntax may look strange at first, but if you understand the fundamentals of lists and for-loops you have all the tools you need to understand list comprehensions![10]

Recall the function doubler() from Listing 7.21 that doubled the elements of a list. This doubling was done “in place” such that the list provided as an argument to the function had its elements doubled. Now we want to write a function called newdoubler() that, like doubler(), doubles every element of the list provided as an argument. However, newdoubler() does not modify the original list argument. Instead, it returns a new list whose elements are double

[10] Despite the elegance and utility of list comprehensions, they will not be used in the remainder of this book and hence the material in this section is not directly relevant to any material in the subsequent chapters.

those of the list argument while the elements of the original list are unchanged. (You can think of the "new" of newdoubler() as being indicative of the fact that this function returns a *new* list.) Listing 7.31 provides the code for newdoubler() and demonstrates its behavior. The code is discussed further following the listing.

Listing 7.31 A modified version of doubler() from Listing 7.21 that returns a new list.

```
1  >>> def newdoubler(xlist):
2  ...     doublelist = []
3  ...     for x in xlist:
4  ...         doublelist.append(2 * x)
5  ...     return doublelist
6  ...
7  >>> mylist = [1, 2, 3]
8  >>> newdoubler(mylist)
9  [2, 4, 6]
10 >>> doubledlist = newdoubler(mylist)
11 >>> doubledlist
12 [2, 4, 6]
13 >>> mylist
14 [1, 2, 3]
```

In line 1 we see that newdoubler() has a single formal parameter xlist. In line 2, the first statement of the body of the function, doublelist is assigned to the empty list. doublelist serves as an accumulator into which we append the doubled values from xlist. This is accomplished with the for-loop in lines 3 and 4. Finally, after the loop is complete, in line 5 doublelist is returned.

Outside the function, in line 7, mylist is set equal to a list of three integers. In line 8 mylist is passed to newdoubler() and we see the returned list in line 9. The values of this list are double those of mylist. newdoubler() is called again in line 10 and the return value stored as doubledlist. Lines 11 through 14 demonstrates that doubledlist contains the doubled values from mylist.

Now that we understand newdoubler() consider the code shown in Listing 7.32 where we now use list comprehension to obtain the doubled list. The code is discussed following the listing.

Listing 7.32 Demonstration of list comprehension in which a new list is produced where the value of its elements are twice that of an existing list.

```
>>> mylist = [1, 2, 3]
>>> # Use list comprehension to obtain a new list where the values are
>>> # twice that of mylist.
>>> [2 * x for x in mylist]
[2, 4, 6]
>>> def lcdoubler(xlist):
```

```
7  ...     return [2 * x for x in xlist]
8  ...
9  >>> doubledlist = lcdoubler(mylist)
10 >>> doubledlist
11 [2, 4, 6]
12 >>> mylist
13 [1, 2, 3]
```

In line 1 we again assign mylist to a list of three integers. Line 4 provides a list comprehension expression that produces a new list where the values are double those of mylist. (The details of this expression will be considered further below.) Given that we can create a new list this way, in lines 6 and 7 we define the function lcdoubler() that has a single argument xlist. The body of the function is merely a return statement that returns the list produced by the list comprehension expression. Lines 9 through 13 demonstrate that lcdoubler() behaves in the same way as newdoubler() (ref. Listing 7.31).

Both lcdoubler() and newdoubler() accomplish the same task, doubling the elements in a list and returning a new list, but using list comprehension lcdoubler() accomplishes in two lines what newdoubler() did in five! list comprehensions can make code smaller, cleaner, and more readable. Of course, until you become comfortable with the syntax of list comprehensions, you may not find them "more readable" than the for-loop constructs we have previously discussed. But, you don't need to work with list comprehensions for long before you realize they are really just a slight syntactic variation on using accumulators, for-loops, and lists to construct a new list from the elements of an existing list. Now, however, is an appropriate time to point out we are not restricted to constructing the new list from an existing list. As will be shown, the new list can be constructed from *any* iterable (such as a tuple or a string).

Let's take another look at the list comprehension in line 4 of Listing 7.32 which is repeated below:

```
[2 * x for x in mylist]
```

Given the output on line 5, we know that this list comprehension produces a new list with values double that of mylist. Now let's consider the various components of this expression and see how it works.

The syntax for a list comprehension consists of brackets containing an expression followed by a for clause. Although this clause can be followed by any number of additional for and if clauses,[11] we will only consider the simplest form of list comprehension with a single for clause. Recall the pattern we are trying to implement: the creation of a new list based on the values in an existing list (or the values from some other iterable). Assuming this new list has the "placeholder" name of *<accumulator>*, the following is a template for implementing this pattern using a for-loop:

```
<accumulator> = []
for <item> in <iterable>:
```

[11] The initial for clause can actually be preceeded by an if clause to perform a form of filtering. if statements are considered in Chap. 11, but not in the context of list comprehensions.

```
    <accumulator>.append(<expression>)
```

In contrast to this, the following is the template for accomplishing the same thing using list comprehension where everything to the right of the assignment operator (equal sign) is the actual list comprehension:

```
<accumulator> = [<expression> for <item> in <iterable>]
```

Of course, with list comprehension, we aren’t obligated to assign the list to an accumulator. We could, for example, have the list comprehension entered directly at the interactive prompt (such as in line 4 of Listing 7.32), or be part of a return statement (such as in line 7 of Listing 7.32), or appear directly as the argument of a print() statement.

In the list comprehensions in Listing 7.32 we didn’t use explicit indexing, but this is certainly an option just as it is within for-loops. This is illustrated in Listing 7.33. In line 2 a list comprehension is again used to double the values of list, but here explicit indexing is used. As we saw from the previous examples, we do not need to use indexing to accomplish this doubling. However, in lines 4 and 5 we define a function called scaler() that “scales” each element of a list based on its index (elements are scaled by the sum of their index plus one). Lines 7 through 10 demonstrate that this function returns the proper result whether passed a list of numbers or strings.

Listing 7.33 Demonstration of the use of explicit indexing within list comprehensions.

```
1  >>> mylist = [1, 2, 3]
2  >>> [2 * mylist[i] for i in range(len(mylist))]
3  [2, 4, 6]
4  >>> def scaler(xlist):
5  ...     return [(i + 1) * xlist[i] for i in range(len(xlist))]
6  ...
7  >>> scaler(mylist)
8  [1, 4, 9]
9  >>> scaler([’a’, ’b’, ’c’, ’d’])
10 [’a’, ’bb’, ’ccc’, ’dddd’]
```

7.8 Chapter Summary

One for-loop can be nested inside another.

One list can be nested inside another. Elements of the outer list are specified by one index enclosed in brackets. An element of an interior list is specified by two indices enclosed in two pairs of brackets. The first index specifies the outer element and the second element specifies the inner element. For example, if x corresponds to the list [[1, 2], [’a’, ’b’, ’c’]], then x[1] corresponds to [’a’, ’b’, ’c’] and x[1][2] corresponds to ’c’.

Nesting can be done to any level. Specification of an element at each level requires a separate index.

Simultaneous assignment can be used to assign values to multiple loop variables in the header of a `for`-loop.

`list`s are mutable, i.e., the elements of a `list` can be changed without creating a new `list`.

When a `list` is assigned to a variable, the variable becomes a reference (or alias) to the `list`. If one `list` variable is assigned to another, both variables point to the same underlying `list`. When a `list` is changed, it is changed globally.

Strings can be used as the iterable in a `for`-loop header in which case the loop variable is assigned the individual characters of the string.

Negative indexing can be used to specify an element of a sequence (e.g., a string, `list`, or `tuple`). The last element of a sequence has an index of `-1`. The first element of a sequence `s` has an index of `-len(s)`.

Slicing is used to obtain a portion of a sequence. For a sequence `s` a slice is specified by

```
s[<start> : <end>]
```

or

```
s[<start> : <end> : <inc>]
```

The resulting sequence takes elements from `s` starting at an index of `<start>` and going up to, but not including, `<end>`. The increment between successive elements is `<inc>` which defaults to `1`. The start and end values can be specified with either positive or negative indexing. `<start>` defaults to `0`. `<end>` defaults to the length of the sequence. `inc` may be negative. When it is negative, the default `<start>` is `-1` and the default `<end>` is `-len(s) - 1`. Reversal of a sequence can be achieved using `[ : : -1]`, and a `list`, e.g., `xlist`, can be copied using `xlist[ : ]`.

7.9 Review Questions

1. What output isproduced by the following code?

```
xlist = [1, [1, 2], [1, 2, 3]]
print(xlist[1])
```

2. What output is produced by the following code?

```
xlist = [1, [1, 2], [1, 2, 3]]
print(xlist[1][1])
```

3. What output is produced by the following code?

```
xlist = [1, [1, 2], [1, 2, 3]]
print(xlist[1] + [1])
```

4. What output is produced by the following code?

```
def sum_part(xlist, n):
    sum = 0
    for x in xlist[n]:
        sum = sum + x
    return sum

ylist = [[1, 2], [3, 4], [5, 6], [7, 8]]
x = sum_part(ylist, 2)
print(x)
```

5. Assume `xlist` is a `list` of `lists` where the inner `lists` have two elements. The second element of these inner `lists` is a numeric value. Which of the following will sum the values of the second element of the nested `lists` and store the result in `sum`?

(a)
```
sum = 0
for item in xlist:
    sum = sum + item[1]
```

(b)
```
sum = 0
for one, two in xlist:
    sum = sum + two
```

(c)
```
sum = 0
for i in range(len(xlist)):
    sum = sum + xlist[i][1]
```

(d) All of the above.

6. What output is produced by the following code?

```
for i in range(3):
    for j in range(3):
       print(i * j, end="")
```

(a) `123246369`

(b) `0000012302460369`

(c) `000012024`

(d) None of the above.

7. What output is produced by the following code?

```
s = "abc"
for i in range(1, len(s) + 1):
    sub = ""
    for j in range(i):
       sub = s[j] + sub
    print(sub)
```

(a)
```
a
ba
cba
```

(b)
```
a
ab
abc
```

(c)
```
a
ab
```

(d) This code produces an error.

8. What output is produced by the following code?

```
s = "grasshopper"
for i in range(1, len(s), 2):
    print(s[i], end="")
```

(a) `gasopr`

(b) `gr`

(c) `rshpe`

(d) `rshper`

9. What output is produced by the following code?

```
x = [7]
y = x
x[0] = x[0] + 3
y[0] = y[0] - 5
print(x, y)
```

10. What output is produced by the following code?

```
x = [7]
y = x
x = [8]
print(x, y)
```

11. What output is produced by the following code?

```
x = [1, 2, 3, 4]
y = x
y[2] = 0
z = x[1 : ]
x[1] = 9
print(x, y, z)
```

12. What output is produced by the following code?

```
s = "row"
for i in range(len(s)):
    print(s[ : i])
```

(a)
```

r
ro
```

(b)
```
r
ro
row
```

(c)
```
ro
row
```

(d) None of the above.

13. What output is produced by the following code?

```
s = "stab"
for i in range(len(s)):
    print(s[i : 0 : -1])
```

(a)
```
s
ts
ats
bats
```

(b)
```

t
at
bat
```

(c)
```

s
st
sta
```

(d) None of the above.

14. What output is produced by the following code?

```
s = "stab"
for i in range(len(s)):
    print(s[i : -5 : -1])
```

(a)
```
s
ts
ats
bats
```

(b)
```

t
at
bat
```

(c)
```

s
st
sta
```

(d) None of the above.

15. What output is produced by the following code?

```
s = "stab"
for i in range(len(s)):
    print(s[0 : i : 1])
```

(a)
```
s
ts
ats
bats
```

(b)
```

t
at
bat
```

(c)
```

s
st
sta
```

(d) None of the above.

ANSWERS: 1) `[1, 2]`; 2) `2`; 3) `[1, 2, 1]`; 4) `11`; 5) d; 6) c; 7) a; 8) c; 9) `[5] [5]`; 10) `[8] [7]`; 11) `[1, 9, 0, 4] [1, 9, 0, 4] [2, 0, 4]`; 12) a; 13) b; 14) a; 15) c.