Chapter 6

Lists and `for`-Loops

To facilitate solving various problems, we often organize data in some form of *data structure*. There are several different kinds of data structures used in computer science. The basic concepts behind some of these structures are already quite familiar to you. For example, there is a structure known as a *stack* that is similar to a stack of plates or trays, but instead of plates we work with data that is only added or removed from the top of the stack. Another data structure is a *queue* which is similar to a line at a checkout counter, but instead of customers, we work with data that is added to the back of the queue but removed from the front of the queue. In contrast to stacks and queues, other ways of organizing data, such as in a binary *tree*, probably aren't familiar to you. The reason there are different types of data structures is that there is no single "best" data structure: one structure may be ideal for solving one type of problem but poorly suited for solving problems of a different type.

In this chapter we introduce what is perhaps the simplest data structure but also arguably the most important, a *list*.[1] You are already well acquainted with lists and have certainly been using them for most of your life. Perhaps you've written "to do" lists, grocery lists, invitation lists, or wish lists. You've seen lists of capitals, countries, colleges, and courses. Lists may be organized in a particular order, such as a top-ten list, or in no particular order, such as the list of nominees for the Academy Awards. Clearly, lists are a part of our daily lives.

When we create a list, it serves to collect the associated data into one convenient "place." There is the list as a whole and then there are the individual items in the list which we typically refer to as the *elements* of the list. In Python, a list is an object with its own class, the `list` class. Thus, we will typically use `Courier` font when we refer to a `list` in the Python sense of the word. Since a `list` is an object, there are various methods that come with it. We will explore a few of these in this chapter.

In the implementation of algorithms, it is quite common that certain statements need to be repeated multiple times. Thus, computer languages invariably provide ways to construct *loops*. In this chapter we will also introduce one such construct: a `for`-loop. In Python a `for`-loop is a form of *definite loop*, meaning that the number of passes through the loop can be determined in advance (or, said another way, the number of times the loop will execute is known *a priori*). So, for example, we might write a `for`-loop to do something with each element in a `list` of five items. We thus know the loop will execute five times. On the other hand, as we will see in Sec. 11.6, there

From the file: `lists-n-loops.tex`

[1]In some languages, what we call a list is called an *array*.

is a different kind of construct known as an *indefinite loop*. The number of times an indefinite loop will execute is typically *not* known in advance. Indefinite loops can be used, for example, to allow a user to enter values until he or she indicates there are no more values to enter.

In many situations, `lists` and `for`-loops go together. A `for`-loop provides a convenient way to process the data contained in a `list`; hence `lists` and `for`-loops are presented together in this chapter. We start by introducing `lists` and follow with a discussion of `for`-loops.

6.1 lists

In Python a `list` is created when comma-separated expressions are placed between square brackets. For example, `[1, "two", 6 / 2]` is a `list`. This `list` has three elements: the first is an integer (`1`); the second is a string (`"two"`); and the third is a `float` since `6 / 2` evaluates to `3.0`. Note that Python will evaluate each expression to determine the value of each element of the `list`. A `list` can be either homogeneous, containing only one type of data, or inhomogeneous, containing different types of data. The example above is inhomogeneous since it contains an integer, a string, and a `float`. (Since a `list` is a form of data, a `list` can, in fact, contain another `list` as one of its elements!)

`lists` can be assigned to a variable or returned by a function. This is demonstrated in Listing 6.1 where a `list` is assigned to `x` in line 1. The `print()` statement in line 2 shows the result of this assignment. The function `f()`, defined in lines 4 and 5, returns a `list` that contains the first four multiples of the parameter passed to the function (actually, as described in more detail below, this is something of an understatement of what `f()` can do).

Listing 6.1 `lists` can be assigned to a variable and returned by a function.

```
1  >>> x = [1, "two", 6 / 2]
2  >>> print(x)
3  [1, 'two', 3.0]
4  >>> def f(n):
5  ...     return [n, 2 * n, 3 * n, 4 * n]
6  ...
7  >>> f(2)
8  [2, 4, 6, 8]
9  >>> y = f(49)
10 >>> type(y)
11 <class 'list'>
12 >>> print(y)
13 [49, 98, 147, 196]
14 >>> f('yo')
15 ['yo', 'yoyo', 'yoyoyo', 'yoyoyoyo']
```

In line 8 we see the `list` produced by the function call `f(2)` in line 7. In line 9 the `list` returned by `f(49)` is assigned to the variable `y`. Lines 10 and 11 show that `y` has a type of `list`. Then, in line 12, a `print()` statement is used to display `y`.

In line 14 `f()` is called with a string argument. Although when `f()` was written we may have had in mind the generation of multiples of a number, we see that `f()` can also be used to produce different numbers of repetition of a string because of operator overloading.

Listing 6.2 provides another example of the creation of a `list`. In lines 2 and 3 the `floats` `a` and `b` are created. In line 4 the list `x` is created for which each element is an expression involving `a` and/or `b`. Line 5 is used to display the contents of `x`. The discussion continues below the listing.

Listing 6.2 Another demonstration of the creation of a `list`. Here we see that after a `list` has been created, subsequent changes to the variables used in the expressions for the elements do not affect the values of the `list`.

```
 1  >>> # Create a list x that depends on the current values of a and b.
 2  >>> a = -233.1789
 3  >>> b = 4.268e-5
 4  >>> x = [a, b, a + b, a - b]
 5  >>> x
 6  [-233.1789, 4.268e-05, -233.17885732, -233.17894268]
 7  >>> # Modify a and b and then confirm that this does not affect x.
 8  >>> a = 1
 9  >>> b = 2
10  >>> x
11  [-233.1789, 4.268e-05, -233.17885732, -233.17894268]
```

In lines 8 and 9 the values of `a` and `b` are changed. The output in line 11 shows these changes to `a` and `b` do not affect the elements of `x`. Once an expression for the element of a `list` has been evaluated, Python will not go back and recalculate this expression (i.e., after the `list` has been created, the `list` is oblivious to subsequent changes to any of the variables that were used in calculating its elements).

6.2 `list` Methods

Because `list` is a class, the objects in this class (or, said another way, the instances of this class) will have various methods and attributes which can be listed using the `dir()` function. Additionally, operator overloading can be used with `lists`. As with strings, we can use the plus sign to concatenate `lists` and the multiplication sign for repetition. The length of a `list` (or `tuple`) can be determined using the built-in function `len()`. Listing 6.3 illustrates these features and also demonstrates the creation of a `list` with no element, i.e., a so-called *empty list* (which we will have occasion to use later).

Listing 6.3 Demonstration of `list` concatenation, the `len()` function, and some of the methods available for `lists`.

```
>>> # Concatenate two lists.
>>> w = ['a', 'bee', 'sea'] + ["solo", "duo", "trio"]
```

```
 3 >>> w
 4 ['a', 'bee', 'sea', 'solo', 'duo', 'trio']
 5 >>> len(w)              # Determine length.
 6 6
 7 >>> dir(w)              # Show methods and attributes.
 8 ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
 9  '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
10  '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
11  '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
12  '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
13  '__reversed__', '__rmul__', '__setattr__', '__setitem__',
14  '__sizeof__', '__str__', '__subclasshook__', 'append', 'count',
15  'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
16 >>> type(w.sort)
17 <class 'builtin_function_or_method'>
18 >>> w.sort()            # Sort list.
19 >>> print(w)
20 ['a', 'bee', 'duo', 'sea', 'solo', 'trio']
21 >>> w.append('quartet') # Append value to list.
22 >>> w
23 ['a', 'bee', 'duo', 'sea', 'solo', 'trio', 'quartet']
24 >>> w.sort()            # Sort list again.
25 >>> w
26 ['a', 'bee', 'duo', 'quartet', 'sea', 'solo', 'trio']
27 >>> z = []              # Create an empty list.
28 >>> print(z)
29 []
30 >>> len(z)
31 0
32 >>> z.append("first")   # Append value to empty list.
33 >>> z
34 ['first']
35 >>> # Extend z with the values from another list.
36 >>> z.extend(["second", "third", "fourth"])
37 >>> z
38 ['first', 'second', 'third', 'fourth']
```

In line 2, two `lists` are concatenated and assigned to `w`. The output on line 4 shows that `w` contains all the elements from the two `lists` that were concatenated. The `len()` function, called on line 5, confirms that `w` is a six-element `list`.

The `dir()` function, called on line 7, shows the attributes and methods for this `list`. Three of the methods (`append()`, `extend()`, and `sort()`) have been highlighted since they are used later in this listing.

In line 16 the `type()` function is used to ascertain the type of `w.sort`. Line 17 reports this is a built-in method. This method is called in line 18 but no output is produced—`sort()` is a void method that returns `None`. However, as you might guess, this method sorts the contents of the `list` on which it is invoked. This is confirmed using the `print()` statement in line 19. In line 20 the elements are in alphabetical order.

A single object can be appended to the end of a `list` using the `append()` method. This is demonstrated in lines 21 through 23. Note that although `w` was sorted alphabetically once, the sorting is not automatically maintained—values appended to the end will simply stay at the end. If a `list` needs to be resorted, this is easily accomplished with another call to the `sort()` method as shown in lines 24 through 26.

In line 27 of Listing 6.3 an empty `list` is created and assigned to `z`. When `z` is printed, we see it has no elements—merely the square brackets are shown. The `len()` function reports the length of `z` is `0`. In line 32 the `append()` method is used to append the string `first` to the list. Since there were no other elements in the list, this string becomes the first and only element. We will often start with an empty `list` and then add elements to it (perhaps the elements are obtained from the user as he or she enters values).

In addition to using the plus sign to concatenate `lists`, the `extend()` method can be used to append the elements of one `list` to another. This is demonstrated in lines 36 through 38 of Listing 6.3.

The way in which `append()` and `extend()` differ is further demonstrated in Listing 6.4. In line 1, the `list` `w` is created and consists of three strings. In line 2 a `list` of two strings is appended to `w`. The `list` is treated as a single object and is made the fourth element of `w`. This is made evident in lines 3 and 4. In line 5 the `list` `u` is created that again consists of three strings. In line 6 `u` is `extend()`'ed by a `list` containing two strings. Lines 7 and 8 show that `u` now contains these two additional strings.

Listing 6.4 Demonstration of the way in which `append()` differs from `extend()`. `append()` appends a single object to a `list` whereas `extend()` appends all the objects from the given iterable to the `list`.

```
1 >>> w = ['a', 'b', 'c']
2 >>> w.append(['d', 'e'])
3 >>> w
4 ['a', 'b', 'c', ['d', 'e']]
5 >>> u = ['a', 'b', 'c']
6 >>> u.extend(['d', 'e'])
7 >>> u
8 ['a', 'b', 'c', 'd', 'e']
```

6.3 for-Loops

The previous section showed some examples of creating a `list`. When we displayed the `lists`, we simply displayed the entire `list`. However, we are often interested in working with the individual elements of a `list`. A `for`-loop provides a convenient way to do this. `for`-loops can be used with more than just `lists`. In Python `lists` are considered a type of *iterable*. An iterable is a data type that can return its elements separately, i.e., one at a time. `for`-loops are, in general, useful when working with any iterable, but here we are mainly concerned with using them with `lists`.

The template for a `for`-loop is shown in Listing 6.5. The header, in line 1, contains the keyword `for` followed by an appropriate lvalue (or identifier) which we identify as *<item>*. For now, you should simply think of *<item>* as a variable which is assigned, with each pass through the loop, the value of each element of the `list`. We will use the term *loop variable* as a synonym for the lvalue *<item>* that appears in the header. The next component of the header is the keyword `in`. This is followed by *<iterable>* which, for now, is taken to mean a `list` or any expression that returns a `list`. As usual, the header is terminated by a colon. The header is followed by indented code that constitutes the body of the loop.[2] The body is executed once for each element of the `list` (or iterable).

Listing 6.5 Template for a `for`-loop.

```
1  for <item> in <iterable>:
2      <body>
```

Another way to interpret a `for`-loop statement is along the lines of: for each element of the `list`, do whatever the body of the loop says to do. You are familiar with this type of instruction since you've heard requests such as, "For each day of the week, tell me what you do." In this case the loop variable is the day. It will take on the values Monday, Tuesday, etc., which come from the list of days contained in "week." For each day you report what you do on that particular day. It may help to think of `for` as meaning "for each."

As something of an aside, we previously said that the body of a function defines a namespace or scope that is unique from the surrounding scope: one can use identifiers defined in the surrounding scope, but any variables defined within the function are strictly local to that function. The body of a function consisted of indented code. The bodies of `for`-loops also consist of indented code. *However*, this is not associated with a separate scope. Any variables defined within the body of a `for`-loop persist in the scope in which the `for`-loop was written.

Let's consider some `for`-loop examples to illustrate their use. In line 1 of Listing 6.6 a `list` of names is created and assigned to the variable `names`. Lines 2 and 3 construct a `for`-loop that iterates through each element of the `names` list. It is a common idiom to have the loop variable be a singular noun, such as `name`, and the `list` variable be a plural noun, such as `names`. However, this is not required and the identifiers used for the `list` and loop variable can be any valid identifier. This is demonstrated in the next loop where, in the header in line 11, the loop variable is `foo`.

Listing 6.6 Demonstrations showing how a `for`-loop can be used to access each of the elements of a `list`.

```
1  >>> names = ["Uma", "Utta", "Ursula", "Eunice", "Unix"]
2  >>> for name in names:
3  ...     print("Hi " + name + "!")
```

[2]When the body of the loop consists of a single line, it may be written on the same line as the header, e.g.,
`for s in ["Go", "Fight", "Win"]: print(s)`
However, we will use the template shown in Listing 6.5 even when the body is a single line.

```
4  ...
5  Hi Uma!
6  Hi Utta!
7  Hi Ursula!
8  Hi Eunice!
9  Hi Unix!
10 >>> count = 0              # Initialize a counter.
11 >>> for foo in names:
12 ...     count = count + 1 # Increment counter.
13 ...     print(count, foo) # Display counter and loop variable.
14 ...
15 1 Uma
16 2 Utta
17 3 Ursula
18 4 Eunice
19 5 Unix
```

The body of the first for-loop consists of a single print() statement as shown in line 3. The argument of this print() statement uses string concatenation to connect "Hi " and an exclamation point with the name that comes from names. We see the output this produces in lines 5 through 9.

In line 10 a counter is initialized to zero. In the following for-loop, this counter is incremented as the first statement in the body, line 12. In line 13 print() is used to display the counter and the loop variable. The loop variable here is foo (again, any valid identifier could have been used). However, in the spirit of readability, it would have been better to have used a more descriptive identifier such as name.

Although the code in Listing 6.6 does not illustrate this, after completion of the loop, the loop variable is still defined and has the value of the last element of the list. So, in this example both name and foo end up with the string value Unix.

6.4 Indexing

As shown in the previous section, for-loops provide a convenient way to sequentially access all the elements of a list. However, we often want to access an individual element directly. This is accomplished by enclosing the *index* of the element in square brackets immediately following the list itself (or a list identifier). The index must be an integer (or an expression that returns an integer). You are undoubtedly used to associating an index of one with the first element of a list. However, this is *not* what is done in many computer languages. Instead, the first element of a list has an index of zero. Although this may seem strange at first, there are compelling reasons for this. In Python (and in C, C++, Java, etc.), you should think of the index as representing the offset from the start of the list. Thus, an index of zero represents the first element of a list, one is the index of the second element, and so on. Pausing for a moment, you may realize that we already have a general way to obtain the index of the last element in the list. This is demonstrated in Listing 6.7.

Listing 6.7 Demonstration of the use of indexing to access individual elements of a `list`.

```
>>> xlist = ["Do", "Re", "Mi", "Fa", "So", "La", "Ti"]
>>> xlist[0]              # First element.
'Do'
>>> xlist[1]              # Second element.
'Re'
>>> xlist[1 + 1]          # Third element.
'Mi'
>>> len(xlist)            # Length of list.
7
>>> xlist[len(xlist) - 1] # Last element.
'Ti'
>>> xlist[len(xlist)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> ["Fee", "Fi", "Fo", "Fum"][1]
'Fi'
>>> ["Fee", "Fi", "Fo", "Fum"][3]
'Fum'
>>> ([1, 2, 3] + ['a', 'b', 'c'])[4]
'b'
```

In line 1 of Listing 6.7 a `list` of seven strings is created and assigned to the variable `xlist`. Lines 2, 4, and 6 access the first, second, and third elements, respectively. Note that in line 6 an expression is used to obtain the index. If the expression evaluates to an integer, this is allowed.

In line 8 the length of the `list` is obtained using the `len()` function. There are seven elements in `xlist`, but the last valid index is one less than this length. In line 10 the index of the last element is obtained by subtracting one from the length. Calculating the index this way for the last element is valid for a `list` of any size.[3][4]

The statement in line 12 shows the error that is produced when the index is outside the range of valid indices, i.e., one obtains an `IndexError`. Keep in mind that the largest valid index is one less than the length of the `list`.

Lines 16 through 19 demonstrate that one can provide an index to directly obtain an index from a `list` literal. Typically this wouldn't be done in practice (why bother to enter the entire `list` into your code if only one element was of interest?). But, this serves to illustrate that one can index any expression that returns a `list`. To truly demonstrate this fact, in line 20 two `lists` are concatenated. This concatenation operation is enclosed in parentheses and produces the new `list` `[1, 2, 3, 'a', 'b', 'c']`. To the right of the parentheses is the integer `4` enclosed in

[3]However, an empty `list` has no valid indices since it has no elements. Thus this approach does not work for calculating the index of the last element of an empty `list`. However, since an empty `list` has no elements, this point is somewhat moot.

[4]As we will see in Sec. 7.5, negative indexing provides a more convenient way to access elements at the end of a `list`.

square brackets. This accesses the fifth element of the list, i.e., this index selects the string `'b'` as indicated by the result shown on line 21.

Let us return to the `for`-loop but now use the loop variable to store an integer instead of an element from a `list`. Listing 6.8 demonstrates that this approach can be used to access the elements of the list `fruits` either in order or in reverse order. The code is further discussed below the listing.

Listing 6.8 Demonstration that the elements of a `list` can be accessed using direct indexing and a `for`-loop. The loop variable is set to an integer in the range of valid indices for the `list` of interest.

```
 1  >>> fruits = ["apple", "banana", "grape", "kiwi", "pear"]
 2  >>> indices = [0, 1, 2, 3, 4]
 3  >>> for i in indices:
 4  ...     print(i, fruits[i])
 5  ...
 6  0 apple
 7  1 banana
 8  2 grape
 9  3 kiwi
10  4 pear
11  >>> for i in indices:
12  ...     index = len(fruits) - 1 - i
13  ...     print(i, index, fruits[index])
14  ...
15  0 4 pear
16  1 3 kiwi
17  2 2 grape
18  3 1 banana
19  4 0 apple
```

In line 1 the list `fruits` is created with five elements. In line 2 the list `indices` is created with elements corresponding to each of the valid indices of `fruits` (i.e., the integers `0` through `4`).

The header of the `for`-loop in line 3 sets the loop variable `i` equal to the elements of `indices`, i.e., `i` will be assigned the values `0` through `4` for passes through the loop. The loop variable `i` is then used in the `print()` statement in line 4 to show the elements of `fruits`. The output in lines 6 through 10 shows both the index `i` and the corresponding element of `fruits`.

The `for`-loop in line 11 again sets `i` to the valid indices of the `fruits` list. However, rather than directly using `i` to access an element of `fruits`, we instead calculate an index that is given by `len(fruits) - 1 - i`. When `i` is zero, this expression yields the index of the last element in `fruits`. When `i` is `1`, this expression yields `3` which is the second to the last element, and so on. Each line of output in lines 15 through 19 shows, in order, the value of `i`, the calculated index, and the element of `fruits` for this index.

6.5 `range()`

In line 2 of Listing 6.8 a `list` called `indices` was created that contains the valid indices for the list `fruits`. We are indeed often interested in accessing each element of a `list` via an index. However, it would be quite inconvenient if we always had to create an `indices` list, as was done in Listing 6.8, that explicitly listed all the indices of interest. Fortunately there is a much better way to obtained the desired indices.

Python provides a function, called `range()`, that generates a sequence of integers. We can use this function to generate the integers corresponding to all the valid indices for a given `list`. Or, as you will see, we can use it to generate some subset of indices. The `range()` function does not actually produce a `list` of integers.[5] However, we can force `range()` to show the entire sequence of values that it ultimately produces if we enclose `range()` in the function `list()`.[6]

The `range()` function is used as indicated in Listing 6.9.

Listing 6.9 The `range()` function and its parameters. Parameters in brackets are optional. See the text for further details.

```
    range([start,] stop [, increment])
```

In its most general form, the `range()` function takes three parameters that we identify as `start`, `stop`, and `increment`. The `start` and `increment` parameters are optional and hence are shown in square brackets.[7] When they are not explicitly provided, `start` defaults to zero and `increment` defaults to positive one. There is no default value for `stop` as this value must always be provided explicitly. When two parameters are given, they are taken to be `start` and `stop`.

The first integer produced by `range()` is `start`. The next integer is given by `start + increment`, the next is `start + 2 * increment`, and so on. `increment` may be positive or negative, so the values may be increasing or decreasing. Integers continue to be produced until reaching the last value "before" `stop`. If `increment` is positive, the sequence of integers ends at the greatest value that is still strictly less than `stop`. On the other hand, if `increment` is negative, then the sequence ends with the smallest value that is still strictly greater than `stop`.

Admittedly, it can be difficult to understand what a function does simply by reading a description of it (such as given in the previous paragraph). However, a few examples usually help clarify the description. Listing 6.10 provides several examples illustrating the behavior of the `range()` function. The `list()` function is used so that we can see, all at once, the values produced by `range()`.

[5] `range()` returns an *iterable* that can be used in a `for`-loop header. With each pass of the loop, the `range()` function provides the next integer in the sequence.

[6] The built-in function `list()` attempts to convert its argument to a `list`. If the conversion is not possible, an exception is raised (i.e., an error is generated). Thus, as with `int()`, `float()`, and the `str()` function, the `list()` function performs a form of data conversion. The reason we avoid using "`list`" as a variable name is so that we don't mask this function in a manner similar to the way `print()` was masked in Listing 2.16.

[7] This way of indicating optional arguments is fairly common and the square brackets here have nothing to do with `lists`.

Listing 6.10 Examples of the behavior of the `range()` function. The `list()` function is merely used to produce all of `range()`'s output in a single `list`.

```
1  >>> # The first three commands are all identical in that the arguments
2  >>> # provided for the start and increment are the same as the default
3  >>> # values of 0 and 1, respectively.
4  >>> list(range(0, 5, 1))  # Provide all three parameters.
5  [0, 1, 2, 3, 4]
6  >>> list(range(0, 5))     # Provide start and stop.
7  [0, 1, 2, 3, 4]
8  >>> list(range(5))        # Provide only stop.
9  [0, 1, 2, 3, 4]
10 >>> list(range(1, 10, 2)) # Odd numbers starting at 1.
11 [1, 3, 5, 7, 9]
12 >>> list(range(2, 10, 2)) # Even numbers starting at 2.
13 [2, 4, 6, 8]
14 >>> list(range(5, 5))     # No output since start equals stop.
15 []
16 >>> list(range(5, 0, -1)) # Count down from 5.
17 [5, 4, 3, 2, 1]
```

The statements in lines 4, 6, and 8 use three arguments, two arguments, and one argument, respectively. However, these all produce identical results since the `start` and `increment` values are set to the default values of `0` and `1`, respectively. The statement in line 10 produces a `list` that starts at `1` and increases by `2`, i.e., it generates odd numbers but stops at `9` (since this is the largest value in the sequence that is less than the `stop` value of `10`). The statement in line 12 uses the same `stop` and `increment` as in line 10 but now the `start` value is `2`. Thus, this statement produces even numbers but stops at `8`.

As lines 14 and 15 show, `range()` does not produce any values if `start` and `stop` are equal. Finally, the statement in line 16 has a `start` value greater than the `stop` value. No integers would be produced if `increment` were positive; however, in this case the `increment` is negative. Hence the resulting sequence shown in line 17 is descending. As always, the result in line 17 does not include the `stop` value.

Assume an arbitrary `list` is named `xlist`: What integers are produced by `range(len(xlist))`? Do pause for a moment to think about this. We know that `len(xlist)` returns the length of its argument. This value, in turn, serves as the sole argument to the `range()` function. As such, `range()` will start by producing `0` and, incrementing by one, go up to one less than the length of the `list`. These are precisely the valid indices for `xlist`! Since we made no assumptions about the number of elements in `xlist`, we can use this construct to obtain the valid indices for any `list`.

Listing 6.11 demonstrates the use of the `range()` function in the context of `for`-loops. The `for`-loop in lines 1 and 2 uses the `range()` function to generate the integers `0` through `4`. The subsequent loop, in lines 9 and 10, shows how the three-parameter form of the `range()` function can generate these values in reverse order.

Listing 6.11 Demonstration of the use of the `range()` function in the context of `for`-loops. The last four loops show how the elements of a `list` can be conveniently accessed with the aid of the `range()` function.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(4, -1, -1):
...     print(i)
...
4
3
2
1
0
>>> # Create a list of toppings and display in order.
>>> toppings = ["cheese", "pepperoni", "pineapple", "anchovies"]
>>> for i in range(len(toppings)):
...     print(i, toppings[i])
...
0 cheese
1 pepperoni
2 pineapple
3 anchovies
>>> # Have index go in descending order to show list in reverse.
>>> for i in range(len(toppings) - 1, -1, -1):
...     print(i, toppings[i])
...
3 anchovies
2 pineapple
1 pepperoni
0 cheese
>>> # Have loop variable take on values in ascending order, but
>>> # use this to calculate index which yields list in reverse order.
>>> for i in range(len(toppings)):
...     print(i, toppings[len(toppings) - 1 - i])
...
0 anchovies
1 pineapple
2 pepperoni
3 cheese
>>> # Obtain first and third topping (indices 0 and 2).
>>> for i in range(0, len(toppings), 2):
```

```
45 ...     print(i, toppings[i])
46 ...
47 0 cheese
48 2 pineapple
```

After creating the `toppings` list in line 18, the `for`-loop in lines 19 and 20 is used to show the elements of `toppings` in order. Note how the `range()` function is used in the header.

The `for`-loop in lines 27 and 28 used the three-parameter form of the `range()` function to cycle the loop variable `i` from the last index to the first. The `for`-loop in lines 36 and 37 also displays toppings in reverse order, but here the loop variable is ascending. Thus, in line 37, the loop variable is subtracted from `len(toppings) - 1` in order to obtain the desired index.

The `range()` function in the header of the `for`-loop in line 44 has an increment of 2. Thus, only the first and third values are displayed in the subsequent output.

6.6 Mutability, Immutability, and Tuples

A `list` serves as a way to collect and organize data. As shown above, we can append or extend a `list`. But, we can also change the values of individual elements of a `list`. We say that a `list` is `mutable` which merely means we can change it. The mutability of `lists` is demonstrated in Listing 6.12.

Listing 6.12 Demonstration of the mutability of a list.

```
1 >>> x = [1, 2, 3, 4]           # Create list of integers.
2 >>> x[1] = 10 + 2              # Change second element.
3 >>> x                          # See what x is now.
4 [1, 12, 3, 4]
5 >>> x[len(x) - 1] = "the end!"   # Change last element to a string.
6 >>> x
7 [1, 12, 3, 'the end!']
```

A list `x` is created in line 1. In line 2 the second element of the `list` is assigned a new value (that is obtained from the expression on the right). Lines 3 and 4 display the change. In line 5 the last element of `x` is set equal to a string. Note that it does not matter what the previous type of an element was—we are free to set an element to anything we wish.

It is worthwhile to spend a bit more time considering line 2. Line 2 employs the assignment operator. Previously we said that, in statements such as this, the expression to the right of the equal sign is evaluated and then the resulting value is assigned to the *lvalue* to the left of the equal sign. You may have wondered why we didn't just say "assigned to the *variable* to the left of the equal sign" or "to the *identifier* to the left of the equal sign." Line 2 shows us the reason for using lvalue instead of variable. If, in line 2, we had written `x = 10 + 2`, then we would indeed have assigned the value on the right to the variable on the left (and, in this case, `x` would now point to the `int` `12` rather than to the list it was originally assigned). But, instead, in line 2 we have `x[1] = 10 + 2`. In this case the variable `x` is associated with the entire list while `x[1]` is a single element of this list. We can assign a value to this element (or use the element in expressions). Since we can assign a value to it, it can appear to the left of the assignment operator and is thus considered an lvalue. An element from a list is not typically considered a variable.

In Sec. 4.3, in connection with returning multiple values from a function, it was mentioned that the data was returned in a *tuple*. A `tuple` is another data type. Its behavior is quite similar to that of a `list`. To access the elements of a `tuple`, one still uses an index enclosed in square brackets. The first element has an index of zero. The length of a `tuple` is given by the `len()` function. One difference between a `tuple` and a `list` is that a `tuple` is created by enclosing the comma-separated data in parentheses (instead of square brackets as is done for a `list`). So, for example, `(1, "two", 2 + 1)` produces a `tuple` with elements of `1`, `"two"`, and `3`. However, if the comma-separated values do not span more than one line, the parentheses are optional. Listing 6.13 shows some examples pertaining to creating and working with `tuple`s.

Listing 6.13 Demonstration of the use of `tuples`.

```
 1  >>> t = 1, "two", 2 + 1
 2  >>> t
 3  (1, 'two', 3)
 4  >>> type(t)
 5  <class 'tuple'>
 6  >>> for i in range(len(t)):
 7  ...     print("t[", i, "] = ", t[i], sep="")
 8  ...
 9  t[0] = 1
10  t[1] = two
11  t[2] = 3
12  >>> z = ("one",
13  ...      "two",
14  ...      3)
15  >>> print(z)
16  ('one', 'two', 3)
```

In line 1 a `tuple` is created and assigned to the variable `t`. Note that no parentheses were used (even though enclosing the values to the right of the assignment operator in parentheses arguably would make the code easier to read). Line 2 is used to echo the `tuple`. We see the values are now enclosed in parentheses. Python will use parentheses to represent a `tuple` whether or not they were present when the `tuple` was created. Line 4 is used to show that `t`'s type is indeed `tuple`. The `for`-loop in lines 6 and 7 shows that a `tuple` can be indexed in the same way that we indexed a `list`. The statement in lines 12 through 14 creates a `tuple` called `z`. Here, since the statement spans multiple lines, parentheses are necessary.

The one major difference between `lists` and `tuples` is that `tuples` are *immutable*, meaning their values *cannot* be changed. This might sound like it could cause problems in certain situations, but, in fact, there is an easy fix if we ever need to change the value of an element in a `tuple`: we can simply convert the `tuple` to a `list` using the `list()` function. The immutability of a `tuple` and the conversion of a `tuple` to a `list` are illustrated in Listing 6.14.

Listing 6.14 Demonstration of the immutability of a `tuple` and how a `tuple` can be converted to a `list`.

```
>>> t = 'won', 'to', 1 + 1 + 1  # Create three-element tuple.
>>> t
('won', 'to', 3)
>>> t[1] = 2          # Cannot change a tuple.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t = list(t)       # Convert tuple to a list.
>>> type(t)
<class 'list'>
>>> t[1] = 2          # Can change a list.
>>> t
['won', 2, 3]
```

The tuple `t` is created in line 1. In line 4 an attempt is made to change the value of the second element in the `tuple`. Since `tuples` are immutable, this produces the `TypeError` shown in lines 5 through 7.

In line 8 the `list()` function is used to convert the tuple `t` to a `list`. This `list` is reassigned back to the variable `t`. Lines 9 and 10 show that the type of `t` has changed and the remaining lines show that we can now change the elements of this `list`. (When we used `t` as the lvalue in line 8, we lost the original tuple. We could have used a different lvalue, say `tList`, in which case the tuple `t` would still have been available to us.)

The detailed reasons for the existence of both `tuples` and `lists` don't concern us so we won't bother getting into them. We will just say that this relates to the way data is managed in memory and how values can be protected from being overwritten. There are times when a programmer does not want the values in a collection of data to be changed. A `tuple` provides a means of protection, although, as we've seen, with some effort, the tuple can be converted (and copied) to another form and then changed.[8]

6.7 Nesting Loops in Functions

It is possible to have a `for`-loop contained within the body of a function: The `for`-loop is said to be *nested* inside the function. Since a `for`-loop has a body of its own, its body must be indented farther than the statements of the body in which it is nested. Listing 6.15 provides a template for a `for`-loop nested inside a function. As indicated in lines 2 and 5, there can be code both before and after the loop, though neither is required.

[8]But, in fact, the original `tuple` wasn't changed—if another variable references this data, the `tuple` will persist in memory in its original form. The `list` that is created is a *copy* of the `tuple` that will occupy different memory than the `tuple`. As mentioned, if this new `list` is assigned to the same identifier as was used for the `tuple`, then this identifier references this new `list`/new memory. However, the assignment, in itself, does not destroy the original `tuple`.

Listing 6.15 Template for nesting a loop inside a function.

```
def <function_name>(<parameter_list>):
    <function_body_before_loop>
    for <item> in <iterable>:
        <for_loop_body>
    <function_body_after_loop>
```

To help illustrate the use of a `for`-loop inside a function assume a programmer wants to write a function that will prompt the user for an integer N and a `float`. The function should display the first N multiples of the `float` and then return the last multiple. We'll start by considering a couple of ways things can go wrong before showing a correct implementation.

Listing 6.16 shows a broken implementation in which the programmer places the prompt for input inside the body of the `for`-loop. In this case the user is prompted for the `float` in each pass of the loop.

Listing 6.16 Flawed implementation of a function where the goal is to show a specified number of multiples of a number that the user enters.

```
1  >>> def multiples():
2  ...     num_mult = int(input("Enter number of multiples: "))
3  ...     for i in range(1, num_mult + 1):
4  ...         x = float(input("Enter a number: "))
5  ...         print(i, i * x)
6  ...     return i * x
7  ...
8  >>> multiples()
9  Enter number of multiples: 4
10 Enter a number: 7
11 1 7.0
12 Enter a number: 7
13 2 14.0
14 Enter a number: 7
15 3 21.0
16 Enter a number: 7
17 4 28.0
18 28.0
```

The problem with this code is that the `input()` statement is in the body of the `for`-loop (see line 4). The user should be prompted for this value *before* entering the loop. As things stand now, when the user requests N multiples, the user also has to enter the `float` value N separate times.

Listing 6.17 shows another broken implementation. This time the user is properly prompted for the number of multiples and the `float` value prior to the loop. However, the `return` statement in line 6 is indented to the same level as the body of the `for`-loop. Because of this, the `return` statement will terminate the function in the first pass of the loop; once a `return` statement is encountered, a function is terminated. This is evident from the output of the function which is shown in line 11, i.e., there is only one line of output despite the fact that the user requested `4` multiples as shown in line 9. When `multiples()` is invoked again, in line 13, the user enters that `4000` multiples are desired. However, again, only one line of output is produced as shown in line 16. (Note that the values displayed in lines 12 and 17 are the return values of the function that are echoed by the interactive environment—these values are not printed by the function itself.) So, keep in mind that the amount of indentation *is* important!

Listing 6.17 Another flawed implementation of a function where the goal is to show a specified number of multiples of a given value.

```
1  >>> def multiples():
2  ...     num_mult = int(input("Enter number of multiples: "))
3  ...     x = float(input("Enter a number: "))
4  ...     for i in range(1, num_mult + 1):
5  ...         print(i, i * x)
6  ...         return i * x
7  ...
8  >>> multiples()
9  Enter number of multiples: 4
10 Enter a number: 7
11 1 7.0
12 7.0
13 >>> multiples()
14 Enter number of multiples: 4000
15 Enter a number: 7
16 1 7.0
17 7.0
```

Finally, Listing 6.18 shows a proper implementation of the `multiples()` function.

Listing 6.18 Function with statements before and after the nested `for`-loop. This implementation properly obtains input from the user prior to the `for`-loop and returns the desired value after the `for`-loop has ended.

```
1  >>> def multiples():
2  ...     num_mult = int(input("Enter number of multiples: "))
3  ...     x = float(input("Enter a number: "))
4  ...     for i in range(1, num_mult + 1):
5  ...         print(i, i * x)
6  ...     return i * x  # Code after and outside the loop.
7  ...
8  >>> multiples()
9  Enter number of multiples: 4
10 Enter a number: 7
```

```
1 7.0
2 14.0
3 21.0
4 28.0
28.0
```

Later we will learn about other constructs, such as `while`-loops and conditional statements, that also have bodies of their own. All these constructs can be nested inside other constructs. In fact, we can nest one `for`-loop within another as will be discussed in Sec. 7.1. The important syntactic consideration is that the body of each of these is indented relative to its header.

6.8 Simultaneous Assignment with Lists

Simultaneous assignment is discussed in Sec. 2.4. In simultaneous assignment there are two or more comma-separated expressions to the right of the equal sign and an equal number of comma-separated lvalues to the left of the equal sign. As discussed in Sec. 6.6, a comma-separated collection of expressions automatically forms a `tuple` (whether or not it is surrounded by parentheses). Thus, another way to think of the simultaneous assignment operations we have seen so far is as follows: The number of lvalues to the left of the equal sign must be equal to the number of elements in the tuple to the right of the equal sign.

Given the claim in Sec. 6.6 that `lists` and `tuples` are nearly identical (with the exception that `lists` are mutable and `tuples` are not), one might guess that `lists` can be used in simultaneous assignment statements too. This is indeed the case. Listing 6.19 illustrate this.

Listing 6.19 Demonstration that `lists`, like `tuples`, can be used in simultaneous assignment statements. The assignments in lines 1 and 4 involve `tuples` while the ones in lines 7 and 11 involve `lists`. All behave the same way in terms of the actual assignment to the lvalues on the left side of the equal sign.

```
1  >>> x, y, z = 1, 2, 3   # Tuple to right, without parentheses.
2  >>> print(x, y, z)
3  1 2 3
4  >>> r, s, t = (1, 2, 3) # Tuple to right, with parentheses.
5  >>> print(r, s, t)
6  1 2 3
7  >>> a, b, c = [1, 2, 3] # List to right.
8  >>> print(a, b, c)
9  1 2 3
10 >>> xlist = [1, 2, 3]
11 >>> i, j, k = xlist
12 >>> print(i, j, k)
13 1 2 3
```

In line 1 simultaneous assignment is used in which a `tuple` appears to the right of the equal sign. This `tuple` is given without parentheses. Lines 2 and 3 show the result of this assignment.

In line 4 simultaneous assignment is again used with a `tuple` appearing to the right of the equal sign. However, parentheses enclose the elements of the `tuple`. These parentheses have no effect, and the statement in line 4 is functionally identical to the statement in line 1.

In lines 7 and 11 simultaneous assignment is used where now a *list* appears to the right of the equal sign. The output in lines 9 and 13 shows these assignments behave in the same way as the assignments in lines 1 and 4. Thus, for simultaneous assignment it does *not* matter if one is dealing with a `list` or a `tuple`.

We previously saw, in Listing 2.9, that simultaneous assignment can be used to swap the values of two variables. This sort of swapping can be done regardless of the type of data to which the variables point. This is illustrated by the code in Listing 6.20.

Listing 6.20 Demonstration that simultaneous assignment can be used with any data type including entire `list`s and `tuple`s.

```
1  >>> p = ['a', 'b', 'c', 'd']  # Assign a list of strings to p.
2  >>> n = (12, 24)              # Assign a tuple of integers to n.
3  >>> print(p, n)               # Print p and n.
4  ['a', 'b', 'c', 'd'] (12, 24)
5  >>> p, n = n, p               # Simultaneous assignment to swap p and n.
6  >>> print(p, n)               # See what p and n are now.
7  (12, 24) ['a', 'b', 'c', 'd']
8  >>> w = "WoW"                 # Define string.
9  >>> print(p, n, w)            # Print variables.
10 (12, 24) ['a', 'b', 'c', 'd'] WoW
11 >>> p, n, w = w, p, n         # Swap variables.
12 >>> print(p, n, w)            # Print variables again.
13 WoW (12, 24) ['a', 'b', 'c', 'd']
```

In lines 1 and 2 a `list` and a `tuple` are assigned to variables. In line 5 the data assigned to these variables are swapped using simultaneous assignment. In line 8 a string is assigned to a variable. In line 11 the `list`, `tuple`, and string data are swapped among the various variables using simultaneous assignment. The output in line 13 shows the result of the swap.

6.9 Examples

In this section we present three examples that demonstrate the utility of lists, `for`-loops, and the `range()` function. We also introduce the concept of an accumulator.

6.9.1 Storing Entries in a `list`

Assume we want to allow the user to enter a list of data. For now we will require that the number of entries be specified in advance and that the entries will be simply stored as strings (if we want to store integers or `float`s, we can use, as appropriate, `int()`, `float()`, or `eval()` to perform the desired conversion of the input).

Listing 6.21 shows code that is suitable for this purpose. In lines 1 through 6 the function `get_names()` is defined. This function takes a single parameter, `num_names`, which is the number of names to be read. The body of the function starts, in line 2, by initializing the `list` `names` to the empty `list`. This is followed by a `for`-loop where the header is merely used to ensure the body of the loop is executed the proper number of times, i.e., it is executed `num_names` times. In the body of the loop, in line 4, the `input()` function is used to prompt the user for a name. Note how the prompt is constructed by adding one to the loop variable, converting this sum to a string (with the `str()` function), and then concatenating this with other strings. (Lines 9 through 11 show the resulting prompt.) The string entered by the user is stored in the variable `name` and this is appended to the `names` `list` in line 5. Please note the indentation that is used. The `for`-loop is *nested* inside the body of the function and hence the body of the `for`-loop must be indented even farther. Finally, in line 6, the `names` `list` is returned by the function. Importantly, note that the `return` statement is outside the `for`-loop since it is not indented at the same level as the body of the loop. Were the `return` statement in the body of the loop, the loop could not execute more than once—the function would be terminated as soon as the `return` statement was encountered. Nesting of a loop in a function is discussed in Sec. 6.7 while the remainder of the code in Listing 6.21 is discussed following the listing.

Listing 6.21 Function to create a `list` of strings where each string is entered by the user.

```
>>> def get_names(num_names):
...     names = []
...     for i in range(num_names):
...         name = input("Enter name " + str(i + 1) + ": ")
...         names.append(name)
...     return names
...
>>> furry_friends = get_names(3)
Enter name 1: Bambi
Enter name 2: Winnie the Pooh
Enter name 3: Thumper
>>> print(furry_friends)
['Bambi', 'Winnie the Pooh', 'Thumper']
>>> n = int(input("Enter number of names: "))
Enter number of names: 4
>>> princesses = get_names(n)
Enter name 1: Cinderella
Enter name 2: Snow White
Enter name 3: Princess Tiana
Enter name 4: Princess Jasmine
>>> princesses
['Cinderella', 'Snow White', 'Princess Tiana', 'Princess Jasmine']
```

In line 8 the `get_names()` function is called with an argument of `3`. Thus, the user is prompted to enter three names as shown in lines 9 through 11. The `list` that contains these names is returned by `get_names()` and, also in line 11, assigned to the variable `furry_friends`. Line 12 is used to show the contents of `furry_friends`.

Instead of "hardwiring" the number of names, in line 14 the user is prompted to enter the desired number of names. The user enters `4` in line 15 and this value is stored in `n`. The `get_names()` function is called in line 16 with an argument of `n`. Hence, the user is prompted for four names as shown in lines 17 through 20. The output in line 22 shows that the user's input is now contained in the `list` `princesses`.

6.9.2 Accumulators

Often we want to perform a calculation that requires the accumulation of values, i.e., the final value is the result of obtaining contributions from multiple parts. Assume we want to find the value of the following series, where N is some integer value:

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{N}.$$

You may already be familiar with the mathematical representation of this series where one would write:

$$\sum_{k=1}^{N}\frac{1}{k}=1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{N}.$$

The symbol Σ is the Greek letter sigma which we use to stand for "sum." The term below Σ specifies the starting value of the "summation variable." In this case the summation variable is k and it starts with a value of 1. The term above Σ specifies the final value of this variable. In this particular case we haven't yet said explicitly what the final numeric value is. Instead, we write that the final value is N with the understanding that this has to be specified (as an integer) when we actually calculate the series. The expression immediately to the right of Σ is a general expression for the individual terms in the sum—the actual value is obtained by plugging in the value of the summation variable as it varies between the initial and final values.

We can't calculate this series all at once. Instead, we start with the first term in the series. We then add the next term. We store this result and then add the next term, and so on. Thus we *accumulate* the terms until we have added all of them. For example, if N is 4, we start with the first term of 1 ($k = 1$). We add 0.5 ($k = 2$) to obtain 1.5. We add 0.33333... ($k = 3$) to obtain 1.83333...; and then we add 0.25 ($k = 4$) to obtain 2.0833333....

Problems that involve the accumulation of data are quite common. Typically we initialize an identifier to serve as an *accumulator*. This initialization involves setting the accumulator to an appropriate value outside a `for`-loop. Then, in the body of the loop, the desired data contribute to the accumulator as necessary. An accumulator is essentially a variable that is used to accumulate information (or data) with each pass of a loop. When an accumulator is modified, its new value is based in some way on its previous value. The example in Listing 6.21 actually also involves an accumulator, albeit of a different type. In that example we accumulated strings into a list. The accumulator is the list `names` which started as an empty list.

We want to write a function that calculates the series shown above (and returns the resulting value). In this case, for which we are summing numeric values, the accumulator should be a `float` that is initialized to `0.0`. This accumulator should be thought of as the running sum of the terms in the series. Listing 6.22 shows the appropriate code to implement this. The function `calc_series()` takes a single argument which is the number of terms in the series.

Listing 6.22 Function to calculate the series $\sum_{k=1}^{N} = 1/k$. The variable `total` serves as an accumulator.

```
1  >>> def calc_series(num_terms):
2  ...     total = 0.0    # The accumulator.
3  ...     for k in range(num_terms):
4  ...         total = total + 1 / (k + 1) # Add to accumulator.
5  ...     return total   # Return accumulator.
6  ...
7  >>> calc_series(1)     # 1 term in series.
8  1.0
9  >>> calc_series(2)     # 2 terms in series.
10 1.5
11 >>> calc_series(4)     # 4 terms in series.
12 2.083333333333333
13 >>> calc_series(400)   # 400 terms in series.
14 6.5699296911765055
15 >>> calc_series(0)     # No terms in series.
16 0.0
```

The function `calc_series()` is defined in lines 1 through 5. It takes one argument, `num_terms`, which is the number of terms in the series (corresponding to N in the equation for the series given above). In line 2 the accumulator `total` is initialized to zero. The body of the `for`-loop in lines 3 and 4 will execute the number of times specified by `num_terms`. The loop variable `k` takes on the values `0` through `num_terms - 1`. Thus, in line 4, we add `1` to `k` to get the desired denominator. Alternatively, we can implement the loop as follows:

```
        for k in range(1, num_terms + 1):
            total = total + 1 / k
```

Or, using the augmented assignment operator for addition, which was discussed in Sec. 2.8.4, an experienced programmer would likely write:

```
        for k in range(1, num_terms + 1):
            total += 1 / k
```

For our purposes, any of these implementations is acceptable although this final form is probably the one that most clearly represents the original mathematical expression.

6.9.3 Fibonacci Sequence

The Fibonacci sequence starts with the numbers 0 and 1. Then, to generate any other number in the sequence, you add the previous two. In general, let's identify numbers in the sequence as F_n where n is an index that starts from 0. We identify the first two numbers in the sequence as $F_0 = 0$

and $F_1 = 1$. Now, what is the next number in the sequence, F_2? Since a number in the sequences is always the sum of the previous two numbers, F_2 is given by $F_1 + F_0 = 1 + 0 = 1$. Moving on to the next number, we have $F_3 = F_2 + F_1 = 2$. Thinking about this for a moment leads to the following equation for the numbers in the Fibonacci sequence:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

There is a rather elegant way Python can be used to generate the numbers in this sequence. The function `fib()` shown in Listing 6.23 calculates F_n for any value of n. Although elegant, this function is somewhat sophisticated because of the way it employs simultaneous assignment. The function is defined in lines 1 through 5. In line 2 the variables `old` and `new` are defined. These are the first two numbers in the sequence. Assume the user only wants the first number of the sequence, i.e., F_0, so that the argument to `fib()` is `0`. When this is the case, the `for`-loop is not executed and the value of `old` (which is `0`) is returned. Even though this function internally has a "`new`" value corresponding to F_1, it returns the "`old`" value.

In fact, this function always returns the "`old`" value and this corresponds to the desired number in the sequence, i.e., the nth value. The function always calculates the $(n + 1)$th value and stores it in `new`, but ultimately it is not returned.

Listing 6.23 Demonstration of a function to calculate the numbers in the Fibonacci sequence.

```
1  >>> def fib(n):
2  ...     old, new = 0, 1      # Initialize starting values.
3  ...     for i in range(n):
4  ...         old, new = new, old + new
5  ...     return old
6  ...
7  >>> fib(0)
8  0
9  >>> fib(1)
10 1
11 >>> fib(2)
12 1
13 >>> fib(3)
14 2
15 >>> fib(4)
16 3
17 >>> fib(45)
18 1134903170
```

The right side of line 4 has both the current value in the sequence and the expression corresponding to the *next* value in the sequence, i.e., `old + new`. These two values are simultaneously assigned to `old` and `new`. Thus the current value gets assigned to `old` and the "next" value gets assigned to `new`. This is repeated until `old` contains the desired number. The calls to `fib()` in lines 7 through 17 demonstrate that the function works properly.

Of course, one does not need to use simultaneous assignment to implement the Fibonacci sequence (recall that not all computer languages allow simultaneous assignment). Listing 6.24 shows an alternate implementation that does not employ simultaneous assignment. In line 5 a temporary variable has to be used to store the value of `new` so that its value can subsequently be assigned to `old` (i.e., after `old` has been used to update `new` in line 6). Both implementations of the Fibonacci sequence require a bit of thought to be fully understood.

Listing 6.24 An alternative implementation of the Fibonacci sequence that does not use simultaneous assignment.

```
 1  >>> def fib_alt(n):
 2  ...     old = 0
 3  ...     new = 1
 4  ...     for i in range(n):
 5  ...         temp = new
 6  ...         new = old + new
 7  ...         old = temp
 8  ...     return old
 9  ...
10  >>> fib_alt(0)
11  0
12  >>> fib_alt(1)
13  1
14  >>> fib_alt(4)
15  3
16  >>> fib_alt(45)
17  1134903170
```

6.10 Chapter Summary

A `list` is a sequential collection of data. The data can differ in terms of type, i.e., a `list` can be inhomogeneous.

`lists` can be created by enclosing comma-separated expressions in square brackets, e.g., `[2, "t", 1 + 1]`.

An empty `list` has no elements, i.e., `[]` is an empty list.

Two `lists` can be concatenated using the + operator.

Repetition of a `list` can be obtained using the $\star$ operator (where one of the operands is a `list` and the other is an integer).

The **`append()`** method can be used to append its argument to the end of a `list`. The **`extend()`** method can be used to add the elements of the argument `list` to the `list` for which the method is invoked. The **`sort()`** method sorts the elements of a `list` in place, i.e., a new `list` isn't created but rather the original `list` is changed.

An individual element of a `list` can be accessed via an integer index. The index is given in square brackets following the list. The index represents an offset from the first element; hence the first element has an index of 0, e.g., xlist[1] is the *second* element of xlist.

len(): returns the length of its argument as an integer. When the argument is a list, len() returns the number of elements.

In general, for a list xlist, the last element has an index of len(xlist) - 1.

A **for-loop** uses the following template:

```
for <item> in <iterable>:
    <body>
```

where <item> corresponds to the *loop variable* and is any valid identifier (or lvalue), <iterable> is an object such as a list that returns data sequentially, and the body is an arbitrary number of statements that are indented to the same level.

range(): function used to produce integers. The general form is range(start, stop, inc). The integers that are produced start at start. Each successive term is incremented by inc. The final value produced is the "last" one before stop. Both inc and start are optional and have default values of 1 and 0, respectively. inc may be positive or negative.

Given a list xlist, range(len(xlist)) will produce, in order, all the valid indices for this list.

The range() function can be used as the iterable in the header of a for-loop. This can be done either to produce a counted loop where the loop variable is not truly of interest or to produce the valid indices of a list (in which case the loop variable is used to access the elements of the list).

list(): returns the list version of its argument. (This function can be used to obtain a list containing all the values generated by the range() function. However, in practice, list() is *not* used with range().)

tuples are similar to lists but the elements of a tuple cannot be changed while the elements of a list can be, i.e., tuples are *immutable* while lists are *mutable*.

lists and tuples can be used in simultaneous assignments. They appear on the right side of the equal sign and the number of lvalues to the left of the equal sign must equal the number of elements in the list or tuple.

6.11 Review Questions

1. True or False: The length of a list is given by the length() function.

2. True or False: The index for the first element of a list is 1, e.g., xlist[1] is the first element of the list xlist.

3. What is the output produced by the following code?

```
xlist = []
xlist.append(5)
xlist.append(10)
print(xlist)
```

(a) `[5, 10]`
(b) `[]`
(c) `5, 10`
(d) `5 10`
(e) This produces an error.
(f) None of the above.

4. What is the output produced by the following code?

```
zlist = []
zlist.append([3, 4])
print(zlist)
```

(a) `[3, 4]`
(b) `[[3, 4]]`
(c) `3, 4`
(d) `3 4`
(e) None of the above.

5. What is the value of `xlist2` after the following statement has been executed?

```
xlist2 = list(range(-3, 3))
```

(a) `[-3, -2, -1, 0, 1, 2, 3]`
(b) `[-3, -2, -1, 0, 1, 2]`
(c) `[-2, -1, 0, 1, 2]`
(d) `[-3, 0, 3]`
(e) This produces an error.

6. What is the value of `xlist3` after the following statement has been executed?

```
xlist3 = list(range(-3, 3, 3))
```

(a) `[-3, 0, 3]`
(b) `[-3, 0]`
(c) `[-2, 1]`
(d) This produces an error.

7. What is the value of `xlist4` after the following statement has been executed?

```
xlist4 = list(range(-3))
```

(a) `[]`
(b) `[-3, -2, -1]`
(c) `[-3, -2, -1, 0]`
(d) This produces an error.

8. What is output produced by the following?

```python
xlist = [2, 1, 3]
ylist = xlist.sort()
print(xlist, ylist)
```

(a) `[2, 1, 3] [1, 2, 3]`
(b) `[3, 2, 1] [3, 2, 1]`
(c) `[1, 2, 3] [2, 1, 3]`
(d) `[1, 2, 3] None`
(e) This produces an error.

9. To what value is the variable `x` set by the following code?

```python
def multiply_list(start, stop):
    product = 1
    for element in range(start, stop):
        product = product * element
        return product

x = multiply_list(1, 4)
```

(a) `24`
(b) `6`
(c) `2`
(d) `1`

10. Consider the following function:

```python
def f1(x, y):
    print([x, y])
```

True or False: This function returns a `list` consisting of the two parameters passed to the function.

11. Consider the following function:

```python
def f2(x, y):
    return x, y
```

True or False: This function returns a `list` consisting of the two parameters passed to the function.

12. Consider the following function:

```
def f3(x, y):
    print(x, y)
    return [x, y]
```

True or False: This function returns a `list` consisting of the two parameters passed to the function.

13. Consider the following function:

```
def f4(x, y):
    return [x, y]
    print(x, y)
```

True or False: This function prints a `list` consisting of the two parameters passed to the function.

14. Consider the following function:

```
def f5(x, y):
    return [x, y]
    print([x, y])
```

True or False: This function prints a `list` consisting of the two parameters passed to the function.

15. What output is produced by the following code?

```
xlist = [3, 2, 1, 0]
for item in xlist:
   print(item, end=" ")
```

(a) `3210`

(b) `3 2 1 0`

(c) `[3, 2, 1, 0]`

(d) This produces an error.

(e) None of the above.

16. What output is produced by the following code?

```
a = 1
b = 2
xlist = [a, b, a + b]
a = 0
b = 0
print(xlist)
```

(a) [a, b, a b]+

(b) [1, 2, 3]

(c) [0, 0, 0]

(d) This produces an error.

(e) None of the above.

17. What output is produced by the following code?

```
xlist = [3, 5, 7]
print(xlist[1] + xlist[3])
```

(a) 10

(b) 12

(c) 4

(d) This produces an error.

(e) None of the above.

18. What output is produced by the following code?

```
xlist = ["aa", "bb", "cc"]
for i in [2, 1, 0]:
   print(xlist[i], end=" ")
```

(a) aa bb cc

(b) cc bb aa

(c) This produces an error.

(d) None of the above.

19. What does the following code do?

```
for i in range(1, 10, 2):
    print(i)
```

(a) Prints all odd numbers in the range [1, 9].

(b) Prints all numbers in the range [1, 9].

(c) Prints all even numbers in the range [1, 10].

(d) This produces an error.

20. What is the result of evaluating the expression `list(range(5))`?

    (a) `[0, 1, 2, 3, 4]`

    (b) `[1, 2, 3, 4, 5]`

    (c) `[0, 1, 2, 3, 4, 5]`

    (d) None of the above.

21. Which of the following headers is appropriate for implementing a counted loop that executes 4 times?

    (a) `for i in 4:`

    (b) `for i in range(5):`

    (c) `for i in range(4):`

    (d) `for i in range(1, 4):`

22. Consider the following program:

```
def main():
    num = eval(input("Enter a number: "))
    for i in range(3):
        num = num * 2
    print(num)

main()
```

Suppose the input to this program is 2, what is the output?

(a)
```
2
4
8
```

(b)
```
4
8
```

(c)
```
4
8
16
```

(d) `16`

23. The following fragment of code is in a program. What output does it produce?

```
fact = 1
for factor in range(4):
    fact = fact * factor
print(fact)
```

    (a) `120`
    (b) `24`
    (c) `6`
    (d) `0`

24. What is the output from the following program if the user enters `5`.

```
def main():
    n = eval(input("Enter an integer: "))
    ans = 0
    for x in range(1, n):
        ans = ans + x
    print(ans)

main()
```

    (a) `120`
    (b) `10`
    (c) `15`
    (d) None of the above.

25. What is the output from the following code?

```
s = ['s', 'c', 'o', 'r', 'e']
for i in range(len(s) - 1, -1, -1):
    print(s[i], end = " ")
```

    (a) `s c o r e`
    (b) `e r o c s`
    (c) `4 3 2 1 0`
    (d) None of the above.

26. The following fragment of code is in a program. What output does it produce?

```
s = ['s', 'c', 'o', 'r', 'e']
sum = 0
for i in range(len(s)):
    sum = sum + s[i]
print(sum)
```

(a) `score`

(b) `erocs`

(c) `scor`

(d) `01234`

(e) None of the above.

27. The following fragment of code is in a program. What output does it produce?

```
s = ['s', 'c', 'o', 'r', 'e']
sum = ""
for i in range(len(s)):
    sum =  s[i] + sum
print(sum)
```

(a) `score`

(b) `erocs`

(c) `scor`

(d) `01234`

(e) None of the above.

28. What is the value returned by the following function when it is called with an argument of `3` (i.e., `summer1(3)`)?

```
def summer1(n):
   sum = 0
   for i in range(1, n + 1):
      sum = sum + i
      return sum
```

(a) `3`

(b) `1`

(c) `6`

(d) `0`

29. What is the value returned by the following function when it is called with an argument of `4` (i.e., `summer2(4)`)?

```python
def summer2(n):
   sum = 0
   for i in range(n):
      sum = sum + i
   return sum
```

   (a) `3`
   (b) `1`
   (c) `6`
   (d) `0`

30. Consider the following function:

```python
def foo():
    xlist = []
    for i in range(4):
        x = input("Enter a number: ")
        xlist.append(x)
    return xlist
```

   Which of the following best describes what this function does?

   (a) It returns a list of four numbers that the user provides.
   (b) It returns a list of four strings that the user provides.
   (c) It returns a list of three numbers that the user provides.
   (d) It produces an error.

ANSWERS: 1) False; 2) False; 3) a; 4) b; 5) b; 6) b; 7) a; 8) d; 9) d (the `return` statement is in the body of the loop); 10) False (this is a void function); 11) False (this function returns a `tuple`); 12) True; 13) False (`print()` statement comes after the `return` statement and thus will not be executed); 14) False; 15) b; 16) b; 17) d; 18) b; 19) a; 20) a; 21) c; 22) d; 23) d; 24) b; 25) b; 26) e; 27) b; 28) b; 29) c; 30) b.