Chapter 11

Conditional Statements

So far we have written code that is executed sequentially, i.e., statements are executed in the order in which they are written. However there are times when we need to alter a program's flow of execution so that certain statements are only executed when a condition is met. To accomplish this we use *conditional statements*. For example, a conditional statement can be used to check whether a student's score on an exam is above a given value. If it is, perhaps a particular message is printed; if not, a different message is printed. As a second example, assume some operation must be performed on lines read from a file, but lines that start with a certain character should be ignored (perhaps this character indicates the line is a comment). In this case we use a conditional statement to check the first character of the line and have the program act appropriately.

The simplest form of a conditional statement is an `if` statement. As you will see, in an `if` statement a *test expression* is checked to see if it is "true." If it is, the body of the `if` statement is executed. Conversely, if the test expression is "false," the body of the `if` statement is not executed.

In this chapter we define what we mean by true and false in the context of conditional statements. We also explore many other programming elements related to conditional statements, including Boolean variables, comparison operators, logical operators, and `while`-loops.

11.1 `if` Statements, Boolean Variables, and `bool()`

George Boole was an English mathematician and logician who lived in the 1800's and formalized much of the math underlying digital computers. A *Boolean expression* is an expression in which the terms have one of only two states which we conveniently think of as true or false. Furthermore, a Boolean expression evaluates to only one of two states: true or false. Although you probably aren't acquainted with all the rigors of Boolean expressions, you are already quite familiar with them at some level because you frequently encounter them in your day-to-day life.

As an example of a Boolean expression, assume you are considering whether or not to go to a Saturday matinee. You might boil your decision down to the weather (you rather be outside when the weather is nice) *and* the movie's rating at **`rottentomatoes.com`** (perhaps you only go to movies that have a rating over 80 percent). We can think of your decision as being governed by a Boolean expression. The terms in the expression are "nice weather" and "rating over 80 percent" (again, in a Boolean expression these terms are either true or false). Ultimately you will see a

From the file: `conditionals.tex`

movie if the following Boolean expression evaluates to true and you will not see it if it evaluates to false: *NOT* nice weather *AND* rating over 80 percent. We will return to expressions involving operators such as *NOT* and *AND* in Sec. 11.4. First, however, we want to consider how the outcome of such expressions can control the flow of a program and consider what "true" and "false" are in terms of a Python program.

In honor of George Boole's rigorous study of logical expressions, i.e., expressions in which terms can only be true or false, in many computer languages, including Python, there is a *Boolean data type*. In Python this type is identified as `bool` and there are two `bool` literals: `True` and `False`.[1]

We will return to Boolean variables shortly, but let us first consider `if` statements. The template for an `if` statement is shown in Listing 11.1. The statement consists of a header, shown in line 1, and a body that can contain any number of statements. The statements that constitute the body must be indented to the same level (similar to the bodies of `for`-loops and function definitions). Within the header is a `test_expression`. If the `test_expression` evaluates to `True`, the body is executed. Conversely, if the `test_expression` evaluates to `False`, the body is not executed and program execution continues with the statement following the body.

Listing 11.1 Template for an `if` statement.

```
1 if <test_expression>:
2     <body>
```

The `test_expression` in an `if` statement always evaluates to an object that is considered either `True` or `False`. There is, for instance, no possibility of something being considered "partially true." It might seem that this places rather tight restrictions on what can serve as a valid `test_expression`. However, this is not the case. In fact, we can use any expression as the `test_expression`! But, let's hold this thought and return to Boolean variables.

Listing 11.2 demonstrates the behavior of `if` statements. The `if` statement in lines 1 and 2 has a test expression that is simply the literal `True`. Since this expression simply evaluates to `True`, the body of the `if` statement is executed, resulting in the output shown in line 4. On the other hand, the `if` statement in lines 5 and 6 has a test expression consisting of the literal `False`. Since this evaluates to `False`, the body of the `if` statement, i.e., line 6 is not executed. Thus, in the interactive environment we simply get the interactive prompt back. The discussion of the code continues following the listing.

Listing 11.2 Demonstration of `if` statements using Boolean literals (`True` and `False`) and a Boolean variable.

```
1 >>> if True:
2 ...     print("It's true!")
3 ...
```

[1] Contrast this to, say, `int`s or `str`s where there are effectively an infinite number of possible literals! The number of digits in an integer literal or the number of characters in a string literal are only limited by the computer's memory and our patience.

```
4  It's true!
5  >>> if False:
6  ...     print("It's true!")
7  ...
8  >>> bt = True
9  >>> type(bt)
10 <class 'bool'>
11 >>> if bt:
12 ...     print("bt is a true variable.  Or should I say True?")
13 ...
14 bt is a true variable.  Or should I say True?
```

In line 8 the variable `bt` is set equal to `True`. The `type()` function is used in line 9 to show that `bt` has a type of `bool`. The `if` statement in lines 11 and 12 and the subsequent output in line 14 demonstrate that a variable can be used as the test expression.

`if` statements that employ only literals, such as those in Listing 11.2, are of no practical use: if we already know a test expression is `True`, we simply write the code in the body (i.e., discard the header). On the other hand, if the test expression is known to be `False`, we can do away with the `if` statement all together. There are several ways to obtain practical and meaningful test expressions. In the following section we discuss *comparison operators* and the important role they play in many conditional statements. Before doing this, however, we want to consider a simpler and remarkably powerful way to obtain meaningful test expressions.

Python is able to map every object to either `True` or `False`. Since the result of any expression is an object, this means that Python is able to use *any* expression as the test expression in an `if` statement header!

Python considers the following to be equivalent to `False`: the numeric value zero (whether integer, `float`, or complex), empty objects (such as empty strings, `lists`, or `tuples`), `None`, and `False`. Everything else is considered to be equivalent to `True`. If you are ever unsure whether a particular object (or expression) is considered `True` or `False`, you can use the `bool()` function to obtain the Boolean representation of this object (or the Boolean representation of whatever object the expression evaluates to). The mapping of objects to Boolean values is illustrated in Listing 11.3.

Listing 11.3 Demonstration that all objects can be treated as Boolean values and hence any expression can be used as the "test expression" in an `if` statement's header.

```
1  >>> x = 0; y = 0.0; z = 0 + 0j
2  >>> bool(x), bool(y), bool(z)
3  (False, False, False)
4  >>> x = -1; y = 1.e-10; z = 0 + 1j
5  >>> bool(x), bool(y), bool(z)
6  (True, True, True)
7  >>> x = []; y = [0]; z = "0"
8  >>> bool(x), bool(y), bool(z)
9  (False, True, True)
10 >>> if 1 + 1:
```

```
11 ...     print("Test expression considered True.")
12 ...
13 Test expression considered True.
14 >>> if 6 * 3 - 18:
15 ...     print("Test expression considered True.")
16 ...
17 >>> s = ""
18 >>> if s:
19 ...     print("Nothing to say.")
20 ...
21 >>>
```

In line 1, identifiers are assigned the integer, `float`, and complex representations of zero. In line 2 `bool()` is used to determine the Boolean equivalents of these values. All are considered `False` as reported in line 3. In line 4 these same identifiers are set to non-zero values. Lines 5 and 6 show that the variables are now all considered to be `True`. In line 7 the variables `x`, `y`, and `z` are assigned an empty `list`, a `list` containing the integer `0`, and a string containing the character `'0'`, respectively. In this case, as shown in lines 8 and 9, `x` is considered `False` while `y` and `z` are considered `True`.

The `if` statement headers in lines 10, 14, and 18 serve to demonstrate that *any* expression can be used as the "test expression." In line 10 the test expression evaluates to `2` which is considered `True` (because it is non-zero) and hence the body of the statement is executed. In line 14, `6 * 3 - 18` evaluates to `0` which is considered `False` and hence the body in line 15 is not executed (note that the string argument of the `print()` function in line 15 does *not* reflect the actual state of the test expression: were the test expression `True`, this string would have been printed). In line 18 the test expression consists simply of the variable `s`. Since `s` was initialized to the empty string in line 17, this is considered `False` and the body is not executed.

To further illustrate what is `True` and what is `False`, the code in Listing 11.4 creates a `list` of objects and then sifts through this `list` to identify the objects considered to be `True`. A `list` of objects is created in line 1 and assigned to the variable `items` while in line 2 the variable `truth` is assigned the empty `list`. The `for`-loop starting in line 3 cycles through each item of `items`. The `if` statement in lines 4 and 5 appends an item to the `truth` list if the item is considered `True` (nothing is done with items considered `False`). Lines 7 and 8 show the objects that are considered `True`. (Items in `items` that do not subsequently appear in `truth` are considered `False`.)

Listing 11.4 Further demonstration of the distinction between objects considered `True` and `False`.

```
1 >>> items = [0, 1, 7 / 2, None, False, "hi", (), 7.3, (0, 0)]
2 >>> truth = []
3 >>> for item in items:
4 ...     if item:
5 ...         truth.append(item)
6 ...
7 >>> truth
8 [1, 3.5, 'hi', 7.3, (0, 0)]
```

Now let us consider two examples that exploit the fact that all expressions in Python evaluate to a value that is considered either to be `True` or `False`. These two examples are somewhat challenging: until you become comfortable with what is and isn't considered to be `True`, the following code may seem slightly mysterious. This code actually can be written in a more readable way using the comparison operators described in the next section. Nevertheless, it is worth taking the time to understand it because doing so will put you well on your way to understanding logical constructs in Python.

Assume we are asked to write a function called `remove_repeats()` that takes a `list` of numbers as its only argument. The function returns a new `list` that is a copy of the original `list` but with any consecutive repeats removed. We are told the original `list` will always have at least one element. As examples of the function's behavior, when it is passed `[1, 2, 2, 3]`, it returns `[1, 2, 3]`, i.e., the second `2` is removed. However, when the function is passed `[1, 2, 3, 2]`, it returns `[1, 2, 3, 2]`, i.e., although `2` appears twice in this `list`, it does not appear consecutively.

How do we implement this function? We know we must cycle over the elements of the original `list`; thus we will need to use a `for`-loop. But, do we need to cycle over all the elements? After some thought, we realize the first element in the `list` has no preceding value, so there is nothing to check regarding it (as the first element, it can't be a repeat!). Thus the `for`-loop can start with the second element. Because we are building a new `list`, we need an accumulator into which we append the non-repeated elements. Rather than initializing this accumulator to an empty `list`, we can initialize it to contain the first value from the original `list`. But, which other elements should be appended to the accumulator? If an element is not the same as the preceding one, then it should be appended. This indicates we need an `if` statement. The "trick," it seems, is to come up with the appropriate test expression for the `if` statement. Listing 11.5, which is discussed below, provides the code to implement this function. (Before looking at this, you may want to try to implement your own solution!)

Listing 11.5 A function that returns a new `list` that is a copy of the `list` the function is passed but consecutively repeated numeric values are removed.

```
>>> def remove_repeats(xlist):
...     clean = [xlist[0]]
...     for i in range(1, len(xlist)):
...         if xlist[i] - xlist[i - 1]:
...             clean.append(xlist[i])
...     return clean
...
>>> remove_repeats([1, 2, 2, 3])
[1, 2, 3]
>>> remove_repeats([1, 2, 3, 2])
[1, 2, 3, 2]
>>> ylist = [2, 7, 7, 7, 8, 7, 7, 9, 1.2, 1.2]
>>> remove_repeats(ylist)
[2, 7, 8, 7, 9, 1.2]
```

The function is defined in lines 1 through 6. The header uses the identifier `xlist` for the sole formal parameter. In line 2 the accumulator `clean` is created with a single element corresponding to the first element of `xlist`. The loop variable `i` in the header of the `for`-loop in line 3 takes on values ranging from 1 to the last valid index for `xlist`. Thus, thinking in terms of indices, for the iterations of the loop, `i` takes on the index of the second element of `xlist`, then the third element, and so on, until reaching the end of `xlist`. Now, consider the `if` statement in lines 4 and 5. The test expression is simply the difference of `xlist[i]` and `xlist[i - 1]`, i.e., the preceding element is subtracted from the "current" element. This difference will be zero when the two elements are equal. Because zero is considered to be `False`, the body of the `if` statement will not be executed if the two values are equal! Lines 8 through 14 show the function works properly.

As another challenging problem, assume we must write a function called `find_averages()` that takes a single argument which is a `list`. Each element of this `list` is itself a `list`. The inner `lists` consist of zero or more numerical values, i.e., the number of elements in each of the inner `lists` is not known in advance. The function `find_averages()` must calculate and print the average of each inner `list` that has at least one element. If an inner `list` has no elements, it is simply ignored. The code to accomplish this is shown in Listing 11.6. The code is discussed following the listing.

Listing 11.6 A function to calculate the average of non-empty `lists` that are contained within an outer `list`.

```
1  >>> def find_averages(xlist):
2  ...     for inner in xlist:
3  ...         if inner:
4  ...             total = 0
5  ...             for num in inner:
6  ...                 total = total + num
7  ...             print(total / len(inner))
8  ...
9  >>> zlist = [[20, 15, 40], [], [5, 8, 10, 15, 100], [3.14]]
10 >>> find_averages(zlist)
11 25.0
12 27.6
13 3.14
```

The `find_averages()` function is defined in lines 1 through 7. The `for`-loop starting in line 2 cycles through each element of the `list` the function is passed. These elements are themselves `lists`. Recall that an empty `list` is considered to be `False`. Thus, only a non-empty `list` will make it into the body of the `if` statement in lines 3 through 7. Empty `lists` are effectively ignored. The code in lines 4 through 7 calculates and prints the average. An integer accumulator is initialized to zero in line 4. The `for`-loop in lines 5 and 6 adds all the values to the accumulator. Finally, the `print()` statement in line 7 displays the average. The remaining lines of the listing demonstrate the function works properly.

Let's introduce another built-in function and consider an alternate implementation `find_averages()`. The built-in function `sum()` returns the sum of an iterable consisting of numeric values. This is

demonstrated in Listing 11.7 where, in line 1, `sum()` is passed the `list` consisting of the integers `1`, `2`, and `3`. The subsequent output, on line 2, is the sum of these values. In line 3, the identifier `ylist` is assigned a `list` of five numbers. This list is passed to the `sum()` function in line 4 and the output on line 5 is the sum of these values. Using the `sum()` function the average of a `list` can be obtained with a single expression and this fact is used in line 9 of Listing 11.7 to simplify the implementation of the `find_averages()` function. The remainder of Listing 11.7 shows that this new implementation of `find_averages()` yields the same results as the implementation in 11.6.

Listing 11.7 Another function to calculate the average of non-empty `lists` that are contained within an outer `list`. This function is similar to the one presented in Listing 11.6, but this function uses the built-in `sum()` function.

```
1  >>> sum([1, 2, 3])
2  6
3  >>> ylist = [7.3, 8.4, 9.5, 100, 1000]
4  >>> sum(ylist)
5  1125.2
6  >>> def find_averages(xlist):
7  ...     for inner in xlist:
8  ...         if inner:
9  ...             print(sum(inner) / len(inner))
10 ...
11 >>> zlist = [[20, 15, 40], [], [5, 8, 10, 15, 100], [3.14]]
12 >>> find_averages(zlist)
13 25.0
14 27.6
15 3.14
```

The code in Listing 11.5 uses the fact that zero is considered to be `False` to affect the flow of execution. Alternatively, we can think of the code in Listing 11.5 as exploiting the fact that only non-zero values are considered to be `True`. The code in Listings 11.6 and 11.7 uses the fact that an empty `list` is considered to be `False` to affect the flow of exectution. An alternate way of thinking about this code is that only non-empty `lists` are considered to be `True`.

11.2 Comparison Operators

The fact that Python is able to treat every object (and hence every expression) as either `True` or `False` provides a convenient way to construct parts of many programs. However, we are often interested in making a decision based on a comparison. For example, when it comes to determining whether the weather is "nice" we may want to base our decision on the temperature and wind speed, for example, is the temperature greater than 60 but less than 80 and is the wind speed less than 15 mph? In order to accomplish such comparisons we use *comparison operators*.[2]

[2]In other languages these are often called relational operators.

Like algebraic operators, comparison operators take two operands. You are already familiar with many of the symbols used for comparison operators. For example, the greater-than sign, >, is used to ask the question: Is the operand on the left greater than the operand on the right? If so, the expression evaluates to `True`. If not, the expression evaluates to `False`. In your math classes the greater than sign was typically used to *establish* a relationship. For example, in a math class $x > 5$ often establishes that the value of x must be greater than 5. However, in Python, `x > 5` does not establish that the variable `x` is greater than `5`. Instead, this expression evaluates whether or not `x` is greater than `5`.

The code in Listing 11.8 illustrates the use of the greater-than comparison operator. In line 1 we ask if `5` is greater than `7`. The `False` in line 2 says this is not so. The simultaneous assignment in line 3 sets `x` and `y` to `45` and `-3.0`, respectively. The comparison in line 4 evaluates to `True` because `x` is greater than `y`. The statement in line 6 assigns to the identifier `result` the result of the comparison on the right side of the assignment operator. The expression on the right side produces the Boolean answer to the question: Is `x` greater than the value of `y` plus `50`? Note that all arithmetic operators have higher precedence than the comparison operators, and all comparison operators have equal precedence. Because `y` plus `50` is `47`, which is greater than the value of `x`, the right side of 6 evaluates to `False`. Indeed, we see, in line 8, that `result` has been assigned `False`. The discussion continues following the listing.

Listing 11.8 Demonstration of the greater-than comparison operator.

```
1  >>> 5 > 7                  # Is 5 greater than 7?
2  False
3  >>> x, y = 45, -3.0
4  >>> x > y                  # Is 45 greater than -3.0?
5  True
6  >>> result = x > y + 50 # Is 45 greater than -3.0 + 50?
7  >>> result
8  False
9  >>> if 1 + 1 > 1:
10 ...     print("I think this should print.")
11 ...
12 I think this should print.
13 >>> "hello" > "Bye"      # Comparison of strings.
14 True
15 >>> "AAB" > "AAC"
16 False
```

The header of the `if` statement in line 9 shows that comparison operators can be used as part of the test expression. Here we are evaluating the question: Is one plus one greater than one? Because this evaluates to `True`, the body of the `if` statement is executed which produces the output shown in line 12.

Python can compare more than just numbers. We can, for example, compare strings. When comparing strings, the comparison is based on ASCII values. Since letters in ASCII are in alphabetic order, comparing ASCII values is effectively the same as comparing alphabetic ordering.

For strings comparison, the comparison starts with the first characters and continues until there is a mismatch in the characters or until reaching the end of the shorter string. Keep in mind that uppercase letters come before lowercase letters. So, for example, `'a'` is greater than `'Z'` while `'z'` is greater than `'A'` (in fact, `'z'` is greater than every letter other than itself!). As lines 13 through 16 of Listing 11.8 show, `hello` is greater than `Bye` but `AAB` is not greater than `AAC`. If one wants to ensure case is ignored in a comparison, the string methods `upper()` or `lower()` can be used to create strings that have all the letters in a single case.

Listing 11.9 lists most of the comparison operators. (Two comparison operators are omitted, one of which is the `is` operator introduced in Sec. 7.3. We return to the other omitted operator in Sec. 11.8.)

Listing 11.9 Comparison operators. These operators are *binary operators* in the sense that they require two operands.

`x < y`	Is `x` less than `y`?
`x <= y`	Is `x` less than or equal to `y`?
`x == y`	Is `x` equal to `y`?
`x >= y`	Is `x` greater than or equal to `y`?
`x > y`	Is `x` greater than `y`?
`x != y`	Is `x` not equal to `y`?

Because of keyboard constraints, we must write >= for "greater than or equal to" rather than the symbol you would typically use in a math class, i.e., $\geq$. Similarly, we write <= for "less than or equal to" instead of $\leq$. Because a single equal sign is the symbol for the assignment operator, we must write two equal signs to test for equality of two operands.[3] To test for inequality we must write != instead of the more familiar symbol $\neq$.

Listing 11.10 demonstrates the use of several comparison operators. In line 1 the integer `7` is compared with the `float 7.0`. The result on line 2 indicates these two are equal. *However*, generally one should *avoid* using the equality comparison with `floats`. Keep in mind that `floats` are often approximations to what we *think* are the values. This is illustrated in lines 3 through 9. In line 3 `x` is initialized to `0.1`, i.e., one-tenth. In line 4 the integer `1` is compared to `10` times `x`. The result on line 5 shows, as we would expect, that one and ten times one-tenth are equal. However, line 6 asks if the integer `1` is equal to the sum of ten copies of `x`. Line 7 reports `False`, i.e., as far as the computer is concerned, one and the sum of ten one-tenth's are not equal! Lines 8 and 9 indicate why this is so. This result is a consequence of the binary representation of one-tenth requiring an infinite number of binary digits. However, only 64 bits are used to store the number in the computer. This truncation results in a slight round-off error. Thus, when we sum `0.1` ten times, we obtain a value that is ever so slightly smaller than `1.0`. Nevertheless, this difference is enough to make the comparison `False`. The discussion continues following the listing.

[3] If you never confuse == and = in conditional statements, you will probably be the first programmer in history to accomplish this feat. In some languages, such as C, this error can lead to insidious bugs. Python, however, does catch this error because Python doesn't allow assignment operations in the header of a conditional statement.

Listing 11.10 Demonstration of the use of various comparison operators.

```
1  >>> 7 == 7.0
2  True
3  >>> x = 0.1
4  >>> 1 == 10 * x
5  True
6  >>> 1 == x + x + x + x + x + x + x + x + x + x
7  False
8  >>> x + x + x + x + x + x + x + x + x + x
9  0.9999999999999999
10 >>> 7 != "7"
11 True
12 >>> 'A' == 65
13 False
14 >>> threshold = 5
15 >>> xlist = [1, 2, 3, 4, 5, 6, 7, 8, 9]
16 >>> at_or_above = []
17 >>> for item in xlist:
18 ...     if item >= threshold:
19 ...         at_or_above.append(item)
20 ...
21 >>> at_or_above
22 [5, 6, 7, 8, 9]
```

In line 10 we ask if the integer `7` is *not* equal to the string `'7'`. Integers are never equal to strings, so this is `True`, i.e., they are not equal. To emphasize this point, in line 12 the string `'A'` is compared to the integer `65`. Recall that the ASCII value for the letter `'A'` is `65`, so we might guess that these would be equal. However, as reported in line 13, these objects are not equal.

The code in lines 14 through 19 appends copies of the values from the `list` `xlist` onto the `list` `at_or_above` if the value is greater than or equal to a specified threshold. The threshold is set to `5` in line 14. The `at_or_above` list is initialized to the empty `list` in line 16. The `for`-loop in lines 17 through 19 cycles through each item of `xlist`. If the item is greater than or equal to the threshold, the value is appended to `at_or_above`. We see, in lines 21 and 22, that the `at_or_above` list contains the values from `xlist` that were equal to or greater than the threshold.[4]

`if` statements can be nested inside other `if` statements. Listing 11.11 demonstrates this. In lines 1 and 2 the identifier `creatures` is assigned a `list` that has four elements. Each of these elements is a three-element `list` that contains values representing a species, a name, and an age. In this particular `list` there are two dogs and two humans. The `for`-loop in lines 3

[4] `list` comprehension, which is discussed in Sec. 7.7, allows one to rewrite lines 16 through 19 as a single statement. It was mentioned in Sec. 7.7 that `list` comprehensions could include conditional statements. The appropriate `list` comprehension would be:
`at_or_above = [item for item in xlist if item >= threshold]`

through 10 cycles through each element of the `creatures list`. In the header of the `for`-loop, simultaneous assignment is used to assign values to the variables `species`, `name`, and `age`. Thus, in the first iteration of the loop when we are working with the first element of the `creatures` list, `species` is assigned `dog`, `name` is assigned `Rover`, and `age` is assigned `7`. The discussion of the code continues following the listing.

Listing 11.11 Demonstration of nested `if` statements.

```
1  >>> creatures = [['dog', 'Rover', 7], ['human', 'Fred', 2],
2  ...               ['dog', 'Fido', 1], ['human', 'Sally', 32]]
3  >>> for species, name, age in creatures:
4  ...     if species == 'dog':
5  ...         if age <= 2:
6  ...             print(name, "is just a pup.")
7  ...         if age > 2:
8  ...             print(name, "is all grown up.")
9  ...     if species == 'human':
10 ...         print("Hi " + name + ".")
11 ...
12 Rover is all grown up.
13 Hi Fred.
14 Fido is just a pup.
15 Hi Sally.
```

The `if` statements in lines 4 and 9 allow us to handle dogs and humans differently (if there were other species in the `creatures` list, nothing would be done with them). Within the body of the `if` statement for dogs are two more `if` statements. The `if` statement in lines 5 and 6 serves to recognize a younger dog while the one in lines 7 and 8 announces a mature dog. The `if` statement for humans, in lines 9 and 10, simply greets the person. The output from the loop is shown in lines 12 through 15.

As an example that ties together many of the things we have learned, let's write a function called `add_day()` that takes a two-element `list` as its single argument. Both elements are integers. The first element represents a month and the second represents a day within that month. The month can be between `1` and `12` while the day is between `1` and the maximum number of days for that month (for this example we ignore leap years and assume February has 28 days). The function `add_day()` adds one day to the date it is passed as an argument and returns the resulting date in a new two-element list. The code for this function is given in Listing 11.12 between lines 1 and 18. The first "real" line in the body of the function, line 8, initializes the `tuple days_in_month` to the number of days in each month. Line 9 simply uses simultaneous assignment to give convenient names to the month and day that were passed to the function. The `if` statement in line 12 checks whether the day is less than the number of days in the given month (note that we must subtract `1` from `month` to obtain the correct index, e.g., the number of days in January is given by `days_in_month[0]`). If the day is indeed less than the number of days in the month, the statement in line 13 returns a two-element `list` in which the month is unchanged and the day is incremented by one. Thus, if and when the body of this `if` statement is executed,

the `return` statement in line 13 ensures that control is returned to the point in the program where this function was called and no other statements in this function are executed. However, if this condition is not `True`, i.e., the day is not less than the number of days in the month, then the body of the `if` statement is skipped and execution of statements continues with line 17. The discussion of this code is continued following the listing.

Listing 11.12 Function to calculate the new month and day when the date is advanced by one day (leap years are ignored).

```
1  >>> def add_day(date):
2  ...     """
3  ...     Add a day to the given date.  The date is a two-element list
4  ...     containing the month and the day of the month.  The function
5  ...     returns a new list containing the new date.
6  ...     """
7  ...     # Number of days in each month.
8  ...     days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
9  ...     month, day = date       # Extract month and day from given date.
10 ...     # If there is another day left in month, simply increment the
11 ...     # day.  Place the month and new day in a list and return it.
12 ...     if day < days_in_month[month - 1]:
13 ...         return [month, day + 1]
14 ...     # If we reached this point, we are at the last day of the month.
15 ...     # So, increment the month (ensuring we return to January if
16 ...     # we're at December), and set the day to 1.  Return new date.
17 ...     month = month % 12 + 1 # Increment month.
18 ...     return [month, 1]
19 ...
20 >>> today = [3, 31]     # March 31st.
21 >>> add_day(today)
22 [4, 1]
23 >>> add_day([7, 3])     # July 3rd.
24 [7, 4]
25 >>> add_day([12, 31])   # New Year's Eve, i.e., December 31st.
26 [1, 1]
```

In line 17 the `month` is taken modulo `12`. Thus, if it is December, `month % 12` evaluates to `0`. This value is incremented by `1` and assigned back to `month`. Given that we just "rolled over" to a new month, the day is set to `1`. The `list` returned in line 18 has the `month` that was calculated in line `17` and the day "hardwired" to `1`.

The remaining lines of Listing 11.12 demonstrate that the function works properly. In line 20 the `list today` is defined with the date corresponding to March 31st. When passed this date, `add_day()` returns the date corresponding to April 1st as shown in line 22. In the following two calls to `add_day()` the date is directly specified as a literal argument. We see that July 3rd is incremented to July 4th and December 31st is incremented to January 1st.

11.3 Compound Conditional Statements

In the `add_day()` function in Listing 11.12 there are some statements that should be executed only if the given day is not the last day of the month. On the other hand, there are other statements that should only be executed if the day *is* the last day of the month. In this function an `if` statement coupled with a `return` statement ensures that only one set of statements is executed. There are many instances like this one in which we want either one block of code or another to be executed. However, we generally can't use (or don't want to use) the `return` statement as is done in the `add_day()` function. Instead, we should use an `if-else` statement. Extending this idea, there are times when there are several different blocks of code that we want to select from, i.e., only one of multiple possible blocks should be executed. In such cases we should use an `if-elif-else` statement. We explore both `if-else` and `if-elif-else` statements in this section.

11.3.1 `if-else` Statements

The template for an `if-else` statement is shown in Listing 11.13.

Listing 11.13 Template for an `if-else` statement.

```
1 if <test_expression>:
2     <body_1>
3 else:
4     <body_2>
```

As with an `if` statement, if the `test_expression` in the header evaluates to `True`, then the code immediately following the header is executed, i.e., the code identified as `body_1`. However, if the `test_expression` evaluates to `False`, then the code following the `else` is executed, i.e., `body_2` is executed but `body_1` is skipped. Thus, one body of code is executed but not the other. Listing 11.14 demonstrates the use of `if-else` statements.

Listing 11.14 Demonstration of an `if-else` statement.

```
1 >>> grade = 95
2 >>> if grade > 80:
3 ...     print("Keep up the good work.")
4 ... else:
5 ...     print("Try working harder.")
6 ...
7 Keep up the good work.
```

In line 1 `grade` is assigned the value `95`. In line 2 we ask if the grade is greater than `80`. Since it is, the first body is executed (and the second body is skipped). This results in the output shown in line 7.

As another example, let's consider a function called `test_square()` that tests whether a number is a "perfect square," i.e., if a given number can be formed by the square of an integer.

If a number is a perfect square, the function announces this fact as well as the integer that forms the perfect square. If the number is not a perfect square, this fact is announced. A function to accomplish this is shown in lines 1 through 7 of Listing 11.15. The code is discussed following the listing.

Listing 11.15 Function to test whether a number is a perfect square. An appropriate message is printed that depends on whether or not the number is a perfect square.

```
1  >>> def test_square(num):
2  ...     root = int(num ** 0.5)
3  ...     if num == root * root:
4  ...         print(num, " is a perfect square (",
5  ...             root, " * ", root, ").", sep="")
6  ...     else:
7  ...         print(num, "is not a perfect square.")
8  ...
9  >>> test_square(49)
10 49 is a perfect square (7 * 7).
11 >>> test_square(50)
12 50 is not a perfect square.
13 >>> test_square(622521)
14 622521 is a perfect square (789 * 789).
```

The function header in line 1 has the single parameter of `num`. In line 2 the square root of `num` is calculated (by raising the value to the power `0.5`). This square root is converted to an integer (i.e., any fractional part is discarded) and assigned to the variable `root`. In line 3 we check if the square of `root` is equal to `num`. If it is, the number must be a perfect square and hence the `print()` statement that spans lines 4 and 5 is executed. If the number is not a perfect square, the statement following the `else` in line 6 is executed, i.e., the `print()` statement in line 7.

In line 9 the function is called with an argument of `49`. Because this is a perfect square, the message in line 10 announces the value as a perfect square. Lines 11 and 12 show that `50` is not a perfect square while lines 13 and 14 show that `622521` is.

In many programs we need to write or use functions that test whether something *is* true. These functions are often identified by names that start with `is`. We have actually seen the names of several of these functions (but we have not used them). Looking closely at Listing 5.5, we see that strings have methods with names such as `isalnum()`, `isupper()`, and `isdigit()`. `isalnum()` returns `True` if the string *is* one that consists of only alphanumeric characters (i.e., letters or digits but not punctuation or whitespace) and has at least one alphanumeric character; `isupper()` returns `True` if the string *is* one where all the "cased" characters are uppercase letters and there is at least one cased letter; and `isdigit` returns `True` if the string *is* one that contains only characters corresponding to digits.

Listing 11.16 demonstrates the use of `isalnum()`, `isupper()`, and `isdigit()` (although here we use these methods on strings consisting of a single character). In line 1 a string is assigned to the variable `s` that has a mix of letters, digits, and punctuation. In lines 2 through 4 a `for`-loop cycles over all the characters of `s`. The `isupper()` method is used to test whether the character

is uppercase. If it is, the `print()` statement in line 4 prints the character. Since only the first character in `s` is uppercase, the output of this loop is merely the letter `T` shown in line 6. The loop in lines 7 to 9 is similar to the previous loop except now the `isalnum()` method is used. In this case the output, shown in line 11, consists of all the letters and digits in the string. The final loop, in lines 12 through 14, uses the `isdigit()` method. In this case the output, shown in lines 16 through 18, is the digits in the string.

Listing 11.16 Demonstration of various "is" methods for strings.

```
1  >>> s = "Test 123!!!"
2  >>> for ch in s:          # Find all uppercase letters.
3  ...     if ch.isupper():
4  ...         print(ch)
5  ...
6  T
7  >>> for ch in s:          # Find all alphanumeric characters.
8  ...     if ch.isalnum():
9  ...         print(ch, end=" ")
10 ...
11 T e s t 1 2 3 >>>
12 >>> for ch in s:          # Find all digits.
13 ...     if ch.isdigit():
14 ...         print(ch)
15 ...
16 1
17 2
18 3
```

Now let's return to the testing of perfect squares. Assume we want to write a function called `is_square()` that returns `True` if the argument is a perfect square and returns `False` otherwise. As a first attempt, we can base this new function on the `test_square()` function in Listing 11.15. Thus, we might implement the function as follows:

```
1  def is_square(num):
2      root = int(num ** 0.5)
3      if num == root * root:
4          return True
5      else:
6          return False
```

This implementation is, in fact, correct in that it does exactly what we want. *However*, this implementation is not considered the best one in that there is really no reason for the `if-else` statement! Keep in mind that expressions with comparison operators evaluate to `True` or `False`. Hence a better implementation of this function is the one shown in lines 1 through 3 in Listing 11.17. Note especially line 3 where the `return` statement says to return the value to which the comparison evaluates. The remaining code in the listing demonstrates that this function works properly.

Listing 11.17 A function to test whether a number is a perfect square. The function directly returns the value to which the comparison evaluates.

```
>>> def is_square(num):
...     root = int(num ** 0.5)
...     return num == root * root
...
>>> is_square(49)
True
>>> is_square(50)
False
>>> if is_square(1234321):
...     print("Perfect square.")
... else:
...     print("Not a perfect square.")
...
Perfect square.
```

11.3.2 `if-elif-else` Statements

Using an `if-elif-else` statement we can write programs that execute at most one among multiple bodies of code. The `elif` should be thought of as "else if." The template for an `if-elif-else` statement is shown in Listing 11.18. The statement must start with an `if` header (and its corresponding body). This is followed by any number of `elif` headers (and their bodies). Each `elif` header has its own test expression. The final `else` is optional. When this statement is executed, the first test expression is evaluated. If it is `True`, the first body is executed and the rest of the `if-elif-else` statement is skipped. If the first test expression evaluates to `False`, the first body is skipped and the second test expression is evaluated. If it is `True`, the second body is executed and the rest of the `if-elif-else` statement is skipped. If the second test expression evaluates to `False`, the second body is skipped and the third test expression is evaluated. And so on. If none of the test expressions are `True`, then the body accompanying the `else` part of the statement is executed if it is present. However, since the `else` part is optional, it may be that none of the bodies are executed.

Listing 11.18 Template for an `if-elif-else` statement.

```
if <test_expression_1>:
    <body1>
elif <test_expression_2>:
    <body2>
elif <test_expression_3>:
    <body3>
      .
```

```
        .
        .
else:
    <bodyN>
```

As an example, let's consider the grading of a course in which scores are on a 100-point scale. For this particular course scores are mapped to letter grades as follows:

$$\begin{array}{ll} \text{A} & 100 \geq \text{score} \geq 90 \\ \text{B} & 90 > \text{score} \geq 80 \\ \text{C} & 80 > \text{score} \geq 70 \\ \text{D} & 70 > \text{score} \geq 60 \\ \text{F} & 60 > \text{score} \end{array}$$

A function to implement this grading is shown in Listing 11.19. The function is defined in lines 1 through 11 while the `if-elif-else` statement spans lines 2 through 11. In line 2 a score is checked to see whether it is greater than or equal to `90`. (Although the upper limit on the score is nominally `100`, we do not check this.) If the score is greater than or equal to `90`, the `print()` statement in line 3 is executed. Then the flow of execution drops to the end of the `if-elif-else` statement. As there are no further statements in this function, Python will return to the point in the program where the function was called (as a reminder, this is a void function because it does not return anything). If the score is less than `90`, the comparison in line 4 is made. Note that we only check if the number is greater than or equal to `80`. From the scoring described above, it seems that to establish whether a score is a B we have to establish that the score is both greater than or equal to `80` *and* less than `90`. This is true, but the first part of the statement takes care of any scores that are at or above `90`. Hence, if we have reached line 4 we already know the score is less than `90`. The rest of the function continues similarly. The remainder of the listing, in lines 13 through 20, consists of calls to `show_grade()` that demonstrate that it works properly.

Listing 11.19 Use of an `if-elif-else` statement to map a numeric score to a letter grade.

```
>>> def show_grade(score):
...     if score >= 90:
...         print("A")
...     elif score >= 80:
...         print("B")
...     elif score >= 70:
...         print("C")
...     elif score >= 60:
...         print("D")
...     else:
...         print("F")
...
>>> show_grade(91)
A
>>> show_grade(80)
```

```
B
>>> show_grade(69)
D
>>> show_grade(20)
F
```

The following is a flawed version of the `show_grade()` function. See whether you can anticipate what this function produces when passed an argument of `91`. (The answer is provided below.)

```
def show_flawed_grade(score):
    if score >= 90:
        print("A")
    if score >= 80:
        print("B")
    if score >= 70:
        print("C")
    if score >= 60:
        print("D")
    else:
        print("F")
```

Calling this function with an argument of `91` results in the following:

```
>>> show_flawed_grade(91)
A
B
C
D
```

The problem with this function is that there are three standalone `if` statements and one `if-else` statement. A score of `91` results in each of the comparisons evaluating to `True`. Hence each body is executed except the one associated with the final `else`. There actually is a way to implement the grading function using standalone `if` statements, but we must ensure that a score is between the upper and lower limits for the particular letter grade. We defer further discussion of this until we have presented the operators described in the next section.

11.4 Logical Operators

We started Sec. 11.1 with a description of a Boolean expression taken from daily life, namely the decision to go to the matinee if the following evaluates to true: *not* nice weather *and* rating over 80 percent. Deciding what constitutes "nice" weather may be a complicated one, but let us assume we are thinking Pythonically and have a Boolean variable `nice_weather` that has been appropriately set. Further assume we have a variable `rating` that stores the rating for a particular movie (and perhaps we'll need a `for`-loop to cycle through the ratings of multiple

movies). We can test if the rating is over 80 using an expression such as `rating > 80`. But how do we combine these things to realize our complete Boolean expression? We actually have learned enough that we can construct somewhat awkward code involving nested `if` statements that will, say, call a `go_to_movie()` function if the Boolean expression is true. However, there is a much more elegant way to implement this if we use *logical operators* which are often called *Boolean operators*.

Logical operators are also something with which you are quite familiar. They are simply the operators `and`, `or`, and `not`. Although you probably don't think of it in these terms, `not` is a unary operator that changes its operand from `True` to `False` or, conversely, from `False` to `True`. Both `and` and `or` are binary operators. An expression involving `and` is considered `True` only if both operands evaluate to `True`. An `or` expression is considered `True` if either or both of the operands evaluate to `True`. Of these three logical operators, `not` has the highest precedence, followed by `and`, and finally `or` has the lowest precedence. All these operators have lower precedence than the comparison operators (and, hence, by extension, lower precedence than the arithmetic operators). As always, if you are ever in doubt about operator precedence, use parentheses to ensure the order of operation you want.

Returning to the movie-going Boolean expression, this could be realized with something as simple as the following:

```
if not nice_weather and rating > 80:
  go_to_movie()
```

This is syntactically correct Python code. If you substitute the word "then" for the colon, this almost reads like English! However, before we dive into more details concerning these operators, we must mention that a straight translation from something said in English to a logical expression isn't always so simple. Without some careful thought, such a translation can lead to subtle bugs and we consider one such example in Sec. 11.7.

Listing 11.20 demonstrates the behavior of the logical operators. Here we use Boolean literals as the operands, but keep in mind that in practice the operands can be any expression! Because `not` has the highest precedence, it modifies the value to its right before the value is used by any other operator. This fact is illustrated in lines 5 and 11. Lines 13 and 15 show that we can have multiple `and`'s and `or`'s in an expression. Line 15 and the subsequent result in line 16 demonstrate that `and` has higher precedence than `or`.

Listing 11.20 Demonstration of the `and`, `or`, and `not` logical operators.

```
1  >>> not True
2  False
3  >>> False and True
4  False
5  >>> not False and True
6  True
7  >>> (not False) and True      # Same as previous statement.
8  True
9  >>> True or False
10 True
```

```
>>> not False or True          # Same as: (not False) or True.
True
>>> not (False or True)
False
>>> False and False or True   # Same as: (False and False) or True.
True
>>> False and (False or True)
False
```

Python allows the conversion of Boolean values to integers using `int()`. The integer equivalent of `False` is `0` while the integer equivalent of `True` is `1`. This conversion is not something we generally care about, but we can exploit it to create easily readable *truth tables* that show the outcomes of a logical expression for all possible combinations of the terms in this expression. This is demonstrated in Listing 11.21 which is discussed following the listing.[5]

Listing 11.21 Truth tables can be constructed by cycling over all the possible inputs to a logical function. Here `0` represents `False` and `1` represents `True`.

```
1  >>> booleans = [False, True]
2  >>> for x in booleans:
3  ...     for y in booleans:
4  ...         print(int(x), int(y), "=>", int(x or y))
5  ...
6  0 0 => 0
7  0 1 => 1
8  1 0 => 1
9  1 1 => 1
10 >>> for x in booleans:
11 ...     for y in booleans:
12 ...         print(int(x), int(y), "=>", int(not x and y or x and not y))
13 ...
14 0 0 => 0
15 0 1 => 1
16 1 0 => 1
17 1 1 => 0
```

In line 1 the `list` `booleans` is initialized to the two possible Boolean values. The header of the `for`-loop in line 2 causes the variable `x` to take on these values while the header of the `for`-loop in line 3 causes the variable `y` also to take on these values. The second loop is nested inside the first. The `print()` statement in line 4 displays (in integer form) the value of `x`, the value of `y`, and the result of the logical expression `x or y`. The output is shown in lines 6 through 9, i.e., the first column corresponds to `x`, the second column corresponds to `y`, and the final column corresponds to the result of the logical expression using these values. The nested `for`-loops in lines 10 through

[5]Because `0` and `1` are functionally equivalent to `False` and `True`, we can use these integer values as the elements of the `list` `booleans` in line 1 with identical results.

12 are set up similarly. The only difference is the logical expression that is being displayed. The expression in line 12 appears much more complicated than the previous one, but the result has a rather simple description in English: the result is `True` if either `x` or `y` is `True` but not if both `x` and `y` are `True`.[6]

11.5 Multiple Comparisons

In the `show_grade()` function in Listing 11.19 an `if-elif-else` statement was used to determine the range within which a given score fell. This construct relied upon the fact that we progressively tested to see if a value exceeded a certain threshold. Once the value exceeded that threshold, the appropriate block of code was executed. But, what if we want to directly check if a value falls within certain limits? Say, for example, we want to test if a score is greater than or equal to 80 but less than 90. If a value falls in this range, assume we want to print a message, for example, `This score corresponds to a B.` How should you implement this? Prior to learning the logical operators in the previous section, we would have used nested `if` statements such as:

```
if score >= 80:
    if score < 90:
        print("This score corresponds to a B.")
```

Having learned about the logical operators, we can write this more succinctly as

```
if score >= 80 and score < 90:
    print("This score corresponds to a B.")
```

In the majority of computer languages this is how you would implement this conditional statement. However, Python provides another way to implement this that is aligned with how ranges are often expressed in mathematics. We can directly "chain" comparison operators. So, the code above can be implemented as

```
if 80 <= score < 90:
    print("This score corresponds to a B.")
```

or

```
if 90 > score >= 80:
    print("This score corresponds to a B.")
```

This can be generalized to any number of operators. If `cmp` is a comparison operator and `op` is an operand, Python translates expressions of the form

```
op1 cmp1 op2 cmp2 op3 ... opN cmpN op{N+1}
```

to

[6] This expression is known as the *exclusive or* of `x` and `y`.

```
(op1 cmp1 op2) and (op2 cmp2 op3) ... and (opN cmpN op{N+1})
```

Even though some operands appear twice in this translation, at most, each operand is evaluated once.

Listing 11.22 demonstrates some of the ways that chained comparison can be used. In line 1 the variables `x`, `y`, and `z` are assigned the values `10`, `20`, and `30`, respectively. In line 2 we ask if `x` is less than `y` and `y` is less than `z`. The result of `True` in line 3 shows both these conditions are met. In line 4 we ask if `10` is equal to `x` and if `x` is less than `z`. The answer is again `True`. In line 6 we ask if `99` is greater than `x` but less than `y`. The `False` in line 7 shows `99` falls outside this range. Finally, the expressions in lines 8 and 10 show that we can write expressions that are not acceptable in mathematics but are valid in Python. In line 8 we are asking if `x` is less than `y` and if `x` is less than `z`. Despite the chaining of operators, Python partially decouples the chain and links the individual expressions with the logical `and` (as mentioned above). So, the expression in line 8 is not making a comparison between `y` and `z`. A similar interpretation should be used to understand the expression in line 10.

Listing 11.22 Demonstration of the use of chained comparisions.

```
1  >>> x, y, z = 10, 20, 30
2  >>> x < y < z
3  True
4  >>> 10 == x <= z
5  True
6  >>> x < 99 < y
7  False
8  >>> y > x < z
9  True
10 >>> x < z > y
11 True
```

11.6 `while`-Loops

By now, we are quite familiar with `for`-loops. `for`-loops are definite loops in that we can typically determine in advance how many times the loop will execute. (A possible exception to this is when a `for`-loop is in a function and there is a `return` statement in the body of the loop. Once the `return` statement is encountered, the loop is terminated as well as the function that contained the loop.) However, often we need to have looping structures in which the number of iterations of the loop cannot be determined in advance. For example, perhaps we want to prompt a user for data and allow the user to keep entering data until the user provides some signal that they are done. Rather than signaling when they are done, the user can potentially be asked to start by specifying the amount of data to be entered. In this case we can stick to using a `for`-loop, but this can be quite inconvenient for the user. Rather than using a definite loop, we want to use an *indefinite loop*

in which the number of times it iterates is not known in advance. In Python (and in many other languages) we implement this using a `while`-loop.

The template for a `while`-loop is shown in Listing 11.23. The header in line 1 consists of the keyword `while` followed by a `test_expression` (which can be any valid expression), and a colon. Following the header is an indented body. The `test_expression` is evaluated. If it evaluates to `True`, then the body of the loop is executed. After executing the body, the `test_expression` is evaluated again. While `test_expression` evaluates to `True`, the body of the loop is executed. When the `test_expression` evaluates to `False`, the loop is terminated and execution continues with the statement following the body.

Listing 11.23 Template for a `while`-loop.

```
1 while <test_expression>:
2     <body>
```

As an example of the use of a `while`-loop, let's write code that prompts the user for names (using the `input()` function). When the user is done entering names, they should signal this by merely hitting return. The code to accomplish this is shown in Listing 11.24. In line 1 the variable `prompt` is assigned the string that is used as the prompt (we create this variable since the prompt is used in two different statements). In line 2 `names` is initialized to the empty `list`. In line 3 the user is prompted to enter a name. The user is told to hit the return key when done. We see the user entered the name `Uma` on line 4 which is assigned to the variable `name`. The `while`-loop starts with the header on line 5 that says the body of the loop should be executed provided the variable `name` is considered `True`. Recall that an empty string is considered `False` but everything else is considered `True`. This header is equivalent to

```
while name != "":
```

However, the statement in line 5 is the more idiomatic way of expressing the desired behavior. The body of the `while`-loop consists of two statements: the first appends `name` to the `names list` while the second prompts the user for another name. In lines 9 through 11 we see the other names the user entered. In line 12 the user responded to the prompt by hitting the return key. Thus `name` was set to the empty string, the `test_expression` for the loop evaluated to `False`, and the loop terminated. Back at the interactive prompt in line 13 we echo the `names list` and see that it contains the four names the user entered.

Listing 11.24 Use of a `while`-loop to records names until the user is done.

```
1 >>> prompt = "Enter name [<ret> when done]: "
2 >>> names = []
3 >>> name = input(prompt)
4 Enter name [<ret> when done]: Uma
5 >>> while name:
6 ...     names.append(name)
7 ...     name = input(prompt)
```

```
...
Enter name [<ret> when done]: Ulrike
Enter name [<ret> when done]: Ursula
Enter name [<ret> when done]: Uta
Enter name [<ret> when done]:
>>> names
['Uma', 'Ulrike', 'Ursula', 'Uta']
```

11.6.1 Infinite Loops and `break`

There is something slightly awkward about the code in Listing 11.24. Note that there is one call to `input()` outside the loop (line 3) and another inside the loop (line 7). Both these calls use the same prompt and assign `input()`'s return value to the same variable, so there appears to be a needless duplication of code, but there doesn't seem to be a simple way around this duplication. The first call to `input()` starts the process. If a user enters a name at the first prompt, then the `while`-loop is executed to obtain a `list` with as many additional names as the user cares to enter.

However, there is an alternative construction that is arguably "cleaner." The new implementation relies on the use of a `break` statement. `break` statements are placed inside loops and are almost always embodied within an `if` statement. When a `break` statement is executed, the surrounding loop is terminated immediately (i.e., regardless of the value of the test expression in the header) and execution continues with the statement following the loop. `break` statements can be used with `for`-loops and `while`-loops.

Before showing examples of `break` statements, let us consider *infinite loops*. Infinite loops are loops that theoretically run forever because the text expression in the header of the loop never evaluates to `False`. Such a situation may arise either because the loop was intentionally coded in such a way or because of a coding error. For example, consider the following code where perhaps the programmer intended to print the numbers 0.1, 0.2, $\cdots$, 0.9:

```
1 x = 0.0
2 while x != 1.0:
3     x = x + 0.1
4     print(x)
```

In line 1 `x` is initialized to `0.0`. The header of the `while`-loop in line 2 dictates that the loop should be executed if `x` is not equal to `1.0`. Looking at line 3 in the body of the loop we see that `x` is incremented by `0.1` for each iteration of the loop. Thus it seems the loop should execute 10 times and then stop. However this is not the case. Recalling the discussion of Listing 11.10, we know that, because of the round-off error in `floats`, summing `0.1` ten times does not equal `1.0`. Thus the value of `x` is never equal to `1.0` and hence the loop does not stop of its own accord (we have to terminate the loop either by hitting control-C on a Mac or Linux machine or by typing control-Z followed by a return on a Windows machine).

If a `while`-loop's test expression is initially `True` and there is nothing done in the body of the loop to affect the test expression, the loop is an infinite loop. It is not uncommon to forget to write the code that affects the test expression. For example, assume we want to implement a

`while`-loop that displays a countdown from a given value to zero and then prints `Blast off!` A first attempt to implement this will often be written as

```
count = 10
while count >= 0:
    print(count, "...", sep="")
print("Blast off!")
```

The problem with this code is that `count` is never changed. A correct implementation is

```
count = 10
while count >= 0:
    print(count, "...", sep="")
    count = count - 1
print("Blast off!")
```

An infinite loop is often created intentionally and written as

```
while True:
    <body>
```

The test expression in the header clearly evaluates to `True` and there is nothing in the body of the loop that can affect this. However, the body will typically contain a `break` statement that is within the body of an `if` statement.[7] The existence of the `break` statement is used to ensure the loop is not truly infinite. And, in fact, no loop is truly infinite as a computer will run out of memory or the power will eventually be turned off. Nevertheless, when the test expression in the header of a `while`-loop will not cause the loop to terminate, we refer to the loop as an infinite loop.

Listing 11.25 demonstrates a function that can be used to implement the countdown described above using an infinite loop and a `break` statement. The function `countdown()` is defined in lines 1 through 7. It takes the single argument `n` which is the starting value for the count. The body of the function contains an infinite loop in lines 2 through 6. The loop starts by printing the value of `n` and then decrementing `n`. The `if` statement in line 5 checks if `n` is equal to `-1`. If it is, the `break` statement in line 6 is executed, terminating the loop. Note that a `break` only terminates execution of the loop. It is not a `return` statement. Thus, when we break out of the loop, the next statement to be executed is the `print()` statement in line 7. (Also, if one loop is nested within another and a `break` statement is executed within the inner loop, it will not break out of the outer loop.)

Listing 11.25 Use of an infinite loop to realize a finite "countdown" function.

```
>>> def countdown(n):
...     while True:
...         print(n, "...", sep="")
...         n = n - 1
```

[7] On the other hand, there are some programs that are designed to run continuously whenever your computer is on. For example, your system may continuously run a program which periodically queries a server to see if you have new email. We will not explicitly consider these types of functions.

```
...         if n == -1:
...             break
...     print("Blast off!")
...
>>> countdown(2)
2...
1...
0...
Blast off!
>>> countdown(5)
5...
4...
3...
2...
1...
0...
Blast off!
```

Now let's implement a new version of the code in Listing 11.24 that reads a list of names. In this new version, shown in Listing 11.26, we incorporate an infinite loop and a `break` statement. This new version eliminates the duplication of code that was present in Listing 11.24. In line 1 `names` is assigned to the empty `list`. The loop in lines 2 through 6 is an infinite loop in that the test expression will always evaluate to `True`. In line 3 the user is prompted for a name. If the user does not enter a name, i.e., if `input()` returns an empty string, the test expression of the `if` statement in line 4 will be `True` and hence the `break` in line 5 will be executed, thus terminating the loop. However, if the user does enter a name, the name is appended to `names` and the body of the loop is executed again. In lines 8 through 11 the user enters four names. In line 12 the user does not provide a name which terminates the loop. The `print()` statement in line 13 and the subsequent output on line 14 show the names have been placed in the `names` list.

Listing 11.26 Use of an infinite loop to obtain a `list` of names.

```
>>> names = []
>>> while True:
...     name = input("Enter name [<ret> when done]: ")
...     if not name:
...         break
...     names.append(name)
...
Enter name [<ret> when done]: Laura
Enter name [<ret> when done]: Libby
Enter name [<ret> when done]: Linda
Enter name [<ret> when done]: Loni
Enter name [<ret> when done]:
>>> print(names)
['Laura', 'Libby', 'Linda', 'Loni']
```

11.6.2 continue

There is one more useful statement for controlling the flow of loops. The `continue` statement dictates termination of the current iteration and a return to execution at the top of the loop, i.e., the test expression should be rechecked and if it evaluates to `True`, the body of the loop should be executed again.

For example, assume we again want to obtain a `list` of names, but with the names capitalized. If the user enters a name that doesn't start with an uppercase letter, we can potentially convert the string to a capitalized string ourselves. However, perhaps the user made a typo in the entry. So, rather than trying to fix the name ourselves, let's start the loop over and prompt for another name. The code in Listing 11.27 implements this function and is similar to the code in Listing 11.26. The difference between the two implementations appears in lines 6 through 8 of Listing 11.27. In line 6 we use the `islower()` method to check if the first character of the name is lowercase. If it is, the `print()` in line 7 is executed to inform the user of the problem. Then the `continue` statement in line 8 is executed to start the loop over. This ensures uncapitalized names are not appended to the `names` list. The remainder of the listing demonstrates that the code works properly.

Listing 11.27 A while loop that uses a `continue` statement to ensure all entries in the `names` list are capitalized

```
1  >>> names = []
2  >>> while True:
3  ...     name = input("Enter name [<ret> when done]: ")
4  ...     if not name:
5  ...         break
6  ...     if name[0].islower():
7  ...         print("The name must be capitalized.  Try again...")
8  ...         continue
9  ...     names.append(name)
10 ...
11 Enter name [<ret> when done]: Miya
12 Enter name [<ret> when done]: maude
13 The name must be capitalized.  Try again...
14 Enter name [<ret> when done]: Maude
15 Enter name [<ret> when done]: Mary
16 Enter name [<ret> when done]: mabel
17 The name must be capitalized.  Try again...
18 Enter name [<ret> when done]: Mabel
19 Enter name [<ret> when done]:
20 >>> print(names)
21 ['Miya', 'Maude', 'Mary', 'Mabel']
```

The `continue` statement can be used with `for`-loops as well. Let's consider one more example that again ties together many things we have learned in this chapter and in previous ones. Assume we want to write code that will read lines from a file. If a line starts with the hash symbol (#), it is taken to be a comment line. Comment lines are printed to the output and the rest of the

loop is skipped. Other lines are assumed to consist of numeric values separated by whitespace. There can be one or more numbers per line. For these lines the average is calculated and printed to the output.

To make the example more concrete, assume the following is in the file `data.txt`:

```
1 # Population (in millions) of China, US, Brazil, Mexico, Iceland.
2 1338 312 194 113 0.317
3 # Salary (in thousands) of President, Senator, Representative.
4 400 174 174
```

This file has two comment lines and two lines of numeric values. Line 2 has 5 values (giving the 2011 population in millions for five countries). Line 4 has three values giving the annual salaries (in thousands of dollars) for the President of the United States and members of the Senate and House of Representatives. Keep in mind that our code can make no assumptions about the number of values in a line (except that there must be at least one).

The code to process the file `data.txt` (or any file similar to `data.txt`) is shown in Listing 11.28. The file is opened in line 1. The `for`-loop starting in line 2 cycles through every line of the file. In line 3 the first character of the line is checked to see if it is a hash. If it is, the line is printed and the `continue` statement is used to start the loop over again, i.e., to get the next line of the file. If the line doesn't start with a hash, the `split()` method is used in line 6 to split the values in the line into individual strings which are stored in the `list numbers`. Recall that although the line contains numeric data, at this point in the code the data is in the form of strings. The `for`-loop in lines 8 and 9 cycles through all the numbers in the line, adding them to `total` which was initialized to zero in line 7. The `float()` function is used in line 9 to covert the strings to `floats`. After the total has been obtained, the `print()` statement in line 10 shows the average, i.e., the total divided by the number of values in the line. The result of processing the file is shown in lines 12 through 15.[8,9]

Listing 11.28 Code to process a file with comment lines and "data lines" where the number of items in a data line is not fixed. The numeric values in data lines are averaged.

```
1  >>> file = open("data.txt")
2  >>> for line in file:
3  ...     if line[0] == '#':   # Comment? Echo line and skip rest of loop.
4  ...         print(line, end="")
5  ...         continue
6  ...     numbers = line.split()  # Split line.
7  ...     total = 0               # Initialize accumulator to zero.
8  ...     for number in numbers:  # Sum all values.
9  ...         total = total + float(number)
10 ...     print("Average =", total / len(numbers)) # Print average.
11 ...
```

[8]The numeric averages are a little "messy" and could be tidied using a format string, but we won't bother to do this.

[9]The built-in function `sum()` cannot be used to directly sum the elements in the `list numbers` since this `list` contains strings and `sum()` requires an iterable of numeric values.

```
12 # Population (in millions) of China, US, Brazil, Mexico, Iceland.
13 Average = 391.4634
14 # Salary (in thousands) of President, Senator, Representative.
15 Average = 249.33333333333334
```

11.7 Short-Circuit Behavior

The logical operators `and` and `or` are sometimes referred to as *short-circuit operators*. This has to do with the fact that Python will not necessarily evaluate both operands in a logical expression involving `and` and `or`. Python only evaluates as much as needed to determine if the overall expression is equivalent to `True` or `False`. Furthermore, these logical expressions don't necessarily *evaluate* to the literal Booleans `True` or `False`. Instead, they evaluate to the value of one operand or the other, depending on which operand ultimately determines whether the expression should be considered `True` or `False`. Exploiting the short-circuit behavior of the logical operators is a somewhat advanced programming technique. It is described here for three reasons: (1) short-circuit behavior can lead to bugs that can be extremely difficult to detect if you don't understand short-circuit behavior, (2) the sake of completeness, and (3) as you progress in your programming you are likely to encounter code that uses short-circuit behavior.

Let us first consider the short-circuit behavior of `and`. If the first operand evaluates to `False`, there is no need to evaluate the second operand because `False and`'ed with anything will still be `False`. To help illustrate this, consider the code in Listing 11.29. In line 1 `False` is `and`'ed with a call to the `print()` function. If `print()` is called, we see an output of `Hi`. However, Python doesn't call `print()` because it can determine this expression evaluates to False regardless of what the second operand returns. In line 3 there is another `and` expression but this time the first operand is `True`. So, Python must evaluate the second operand to determine if this expression should be considered `True` or `False`. The output of `Hi` on line 4 shows that the `print()` function is indeed called. But, what does the logical expression in line 3 evaluate to? The output in line 4 is rather confusing. Does this expression evaluate to the string `Hi`? The answer is no, and the subsequent lines of code, discussed below the listing, help explain what is going on.

Listing 11.29 Demonstration of the use of `and` as a short-circuit operator.

```
1  >>> False and print("Hi")
2  False
3  >>> True and print("Hi")
4  Hi
5  >>> x = True and print("Hi")
6  Hi
7  >>> print(x)
8  None
9  >>> if True and print("Hi"):
10 ...     print("The text expression evaluated to True.")
11 ... else:
12 ...     print("The text expression evaluated to False.")
```

```
13 ...
14 Hi
15 The text expression evaluated to False.
```

In line 5 the result of the same logical expression is assigned to the variable `x`. This assignment doesn't change the fact that the `print()` function is called which produces the output shown in line 6. In line 7 we print `x` and see that it is `None`. Where does this come from? Recall that `print()` is a void function and hence evaluates to `None`. For this logical expression Python effectively says, "I can't determine if this logical expression should be considered `True` or `False` based on just the first operand, so I will evaluate the second operand. I will use whatever the second operand returns to represent the value to which this logical expression evaluates." Hence, the `None` that `print()` returns is ultimately the value to which this logical expression evaluates. Recall that `None` is treated as `False`. So, if this (rather odd) logical expression is used in a conditional statement, as done in line 9, the test expression is considered to be `False` and hence only the `else` portion of this `if-else` statement is executed as shown by the output in line 15. The `Hi` that appears in line 14 is the result of the `print()` function being called in the evaluation of the header in line 9.

To further illustrate the short-circuit behavior of `and`, consider the code in Listing 11.30. The short-circuit behavior of `and` boils down to this: If the first operand evaluates to something that is equivalent to `False`, then this is the value to which the overall expression evaluates. If the first operand evaluates to something considered to be `True`, then the overall expression evaluates to whatever value the second operand evaluates. In line 1 of Listing 11.30 the first operand is an empty `list`. Since this is considered to be `False`, it is assigned to `x` (as shown in lines 2 and 3). In line 4 the first operand is considered to be `True`. Thus the second operand is evaluated. In this case the second operand consists of the expression `2 + 2` which evaluates to `4`. This is assigned to `x` (as shown in lines 5 and 6).

Listing 11.30 Using the short-circuit behavior of the `and` operator to assign one value or the other to a variable.

```
1 >>> x = [] and 2 + 2
2 >>> x
3 []
4 >>> x = [0] and 2 + 2
5 >>> x
6 4
```

The short-circuit behavior of the `or` operator is similar to, but essentially the converse of, the short-circuit behavior of `and`. In the case of `or`, if the first operand is effectively `True`, there is no need to evaluate the second operand (since the overall expression will be considered to be `True` regardless of the second operand). On the other hand, if the first operand is considered to be `False`, the second operand must be evaluated. The value to which an `or` expression evaluates is the same as the operand which ultimately determines the value of the expression. Thus, when used in assignment statements, the value of the first operand is assigned to the variable if this first operand is effectively `True`. If it is not, the value of the second operand is used. This is

illustrated in Listing 11.31. In line 1 the two operands of the `or` operator are `5 + 9` and the string `hello`. Both these operands are effectively `True` (since the first operand evaluates to `14` which is non-zero). However, Python never "sees" the second operand because it knows the outcome of the logical expression simply from the first operand. The logical expression thus evaluates to `14` which is assigned to `x` (as shown in lines 2 and 3). Line 4 differs from line 1 in that the first operand is now the expression `5 + 9 - 14`. This evaluates to zero and thus the output of the logical expression hinges on the value to which the second operand evaluates. Thus the value to which the entire logical expression evaluates is the string `hello`.

Listing 11.31 Demonstration of the short-circuit behavior of the `or` operator.

```
1 >>> x = 5 + 9 or "hello"
2 >>> x
3 14
4 >>> x = 5 + 9 - 14 or "hello"
5 >>> x
6 'hello'
```

It may not be obvious where one would want to use the short-circuit behavior of the logical operators. As an example of their utility, assume we want to prompt a user for a name (using the `input()` function), but we also want to allow the user to remain anonymous by simply hitting return at the input prompt. When the user doesn't enter a name, `input()` will return the empty string. Recall that the empty string is equivalent to `False`. Given this, the code in Listing 11.32 demonstrates how we can prompt for a name and provide a default of `Jane Doe` if the user wishes to remain anonymous. In line 1 the `or` operator's first operand is a call to the `input()` function. If the user enters anything, this will be the value to which the logical expression evaluates. However, if the user does not provide a name (i.e., `input()` returns the empty string), then the logical expression evaluates to the default value `Jane Doe` given by the second operand. When line 1 is executed, the prompt is produced as shown in line 2. In line 2 we also see the user input of `Mickey Mouse`. Lines 3 and 4 show that this name is assigned to `name`. Line 5 is the same as the statement in line 1. Here, however, the user merely types return at the prompt. In this case `name` is assigned `Jane Doe` as shown in lines 7 and 8.

Listing 11.32 Use of the short-circuit behavior of the `or` operator to create a default for user input.

```
1 >>> name = input("Enter name: ") or "Jane Doe"
2 Enter name: Mickey Mouse
3 >>> name
4 'Mickey Mouse'
5 >>> name = input("Enter name: ") or "Jane Doe"
6 Enter name:
7 >>> name
8 'Jane Doe'
```

Of course, we can provide a default without the use of a short-circuit operator. For example, the following is equivalent to lines 1 and 5 of Listing 11.32:

```
1 name = input("Enter name: ")
2 if not name:
3     name = "Jane Doe"
```

This code is arguably easier to understand than the implementation in Listing 11.32. However, since it requires three lines as opposed to the single statement in Listing 11.32, many experienced programmers opt to use the short-circuit approach.

Let's assume you have decided not to exploit the short-circuit behavior of the logical operators. However, there is a chance that this behavior can inadvertently sneak into your code. For example, assume you have a looping structure and for each iteration of the loop you "ask" the user if the program should execute the loop again. If the user wants to continue, they should enter a string such as `Yes`, `yep`, or simply `y`, i.e., anything that starts with an uppercase or lowercase y. Any other response means the user does not want to continue. An attempt to implement such a loop is shown in Listing 11.33. The code is discussed after the listing.

Listing 11.33 A flawed attempt to allow the user to specify whether or not to continue executing a `while`-loop. Because of the short-circuit behavior of the `or` operator this is an infinite loop.

```
1  >>> response = input("Continue? [y/n] ")
2  Continue? [y/n] y
3  >>> while response[0] == 'y' or 'Y':    # Flawed test expression.
4  ...     print("Continuing...")
5  ...     response = input("Continue? [y/n] ")
6  ...
7  Continuing...
8  Continue? [y/n] Yes
9  Continuing...
10 Continue? [y/n] No
11 Continuing...
12 Continue? [y/n] Stop!
13 Continuing...
14 Continue? [y/n] Quit!
15 Continuing...
16     .
17     .
18     .
```

In line 1 the user is prompted as to whether the loop should continue (at this point, the prompt is actually asking whether to enter the loop in the first place). Note that the user is prompted to enter `y` or `n`. However, in the interest of accommodating other likely replies, we want to "secretly" allow any reply and will treat replies that resemble yes to be treated in the same way as a reply of `y`. Thus, the test expression in line 3 uses the first character of the response and appears to ask if this letter is `y` or `Y`. *However*, this is not actually what the code does. Recall that logical operators

have lower precedence than comparison operators. Thus, `response[0]` is compared to `y`. If this evaluates to `False`, the `or` operator will return the second operand, i.e., the test expression evaluates to `Y`. As the character `Y` is a non-empty string, it is considered to be `True`. Therefore the test expression is always considered to be `True`! We have implemented an infinite loop. This is demonstrated starting in line 7 and continuing through the end of the listing. We see that even when the user enters `No` or `Stop!` or anything else, the loop continues.

There are various ways to correctly implement the header of the `while`-loop in Listing 11.33. Here is one approach that explicitly compares the first character of the response to `y` and `Y`:

```
while response[0] == 'y' or response[0] == 'Y':
```

Another approach is to use the `lower()` method to ensure we have the lowercase of the first character and then compare this to `y`:

```
while response[0].lower() == 'y':
```

11.8 The `in` Operator

Two comparison operators were omitted from the listing in Listing 11.9: `in` and `is`. The `is` operator is discussed in Sec. 7.3. It returns `True` if its operands refer to the same memory and returns `False` otherwise. The `is` operator can be a useful debugging or instructional tool, but otherwise it is not frequently used. In contrast to this, the `in` operator provides a great deal of utility and is used in a wide range of programs. The `in` operator answers the question: is the left operand contained in the right operand? The right operand must be a "collection," i.e., an iterable such as a `list` or tuple.

We can understand the operation of the `in` operator by writing a function that mimics its behavior. Listing 11.34 defines a function called `my_in()` that duplicates the behavior of the `in` operator. The only real difference between `my_in()` and `in` is that the function takes two arguments whereas the operator is written between two operands. The function is defined in lines 1 through 5. It has two parameters called `target` and `container`. The goal is to return `True` if `target` matches one of the (outer) elements of `container`. (If `container` has other containers nested inside it, these are not searched.) If `target` is not found, the function returns `False`. The body of the function starts with a `for`-loop which cycles over the elements of `container`. The `if` statement in line 3 tests whether the element matches the `target`. (Note that the test uses the "double equal" comparison operator.) If the `target` and element are equal, the body of the `if` statement is executed and returns `True`, i.e., the function is terminated at this point and control returns to the point of the program at which the function was called. If the `for`-loop cylces through all the elements of the container without finding a match, we reach the last statement in the body of the function, line 5, which simply returns `False`. Discussion of this code continues following the listing.

Listing 11.34 The function `my_in()` duplicates the functionality provided by the `in` operator.

```
1 >>> def my_in(target, container):
```

```
...     for item in container:
...         if item == target:
...             return True
...     return False
...
>>> xlist = [7, 10, 'hello', 3.0, [6, 11]]
>>> my_in(6, xlist)         # 6 is not in list.
False
>>> my_in('hello', xlist)   # String is found in list.
True
>>> my_in('Hello', xlist)   # Match is case sensitive.
False
>>> my_in(10, xlist)        # Second item in list.
True
>>> my_in(3, xlist)         # Integer 3 matches float 3.0.
True
```

In line 7 the `list` `xlist` is created with two integers, a string, a `float`, and an embedded `list` that contains two integers. In line 8 `my_in()` is used to test whether `6` is in `xlist`. The integer `6` is contained within the `list` embedded in `xlist`. However, since the search is only performed on the outer elements of the container, line 9 reports that `6` is not in `xlist`. In line 10 we check whether `hello` is in `xlist`. Line 11 reports that it is. Matching of strings is case sensitive as lines 12 and 13 demonstrate. In line 16 we ask whether the integer `3` is in `xlist`. Note that `xlist` contains the `float` `3.0`. Nevertheless, these are considered equivalent (i.e., the == operator considers `3` and `3.0` to be equivalent).

Listing 11.35 defines the same values for `xlist` as used in Listing 11.34. The statements in lines 2 through 11 perform the same tests as performed in lines 8 through 17 of Listing 11.34 except here the built-in `in` operator is used. Line 12 shows how we can check whether a target is *not* in the container. Note that we can write `not in` which is similar to how we express this query in English. However, we can also write this expression as shown in line 14. (The statement in line 12 is the preferred idiom.)

Listing 11.35 Demonstration of the use of the `in` operator.

```
>>> xlist = [7, 10, 'hello', 3.0, [6, 11]]
>>> 6 in xlist
False
>>> 'hello' in xlist
True
>>> 'Hello' in xlist
False
>>> 10 in xlist
True
>>> 3 in xlist
True
>>> 22 not in xlist     # Check whether target is not in container.
```

```
13 True
14 >>> not 22 in xlist     # Alternate way to write previous expression.
15 True
```

To demonstrate one use of `in`, let's write a function called `unique()` that accepts any iterable as an argument. The function returns a `list` with any duplicates removed from the items within the argument. Listing 11.36 defines the function in lines 1 through 6. The argument to the function is the "container" `dups` (which can be any iterable). In line 2 the `list` `no_dups` is initialized to the empty `list`. The `for`-loop in lines 3 through 5 cycles through all the elements of `dups`. The `if` statement in lines 4 and 5 checks whether the element is *not* currently in the `no_dups` `list`. If it is not, the item is appended to `no_dups`. After cycling through all the elements in `dups`, the `no_dups` `list` is returned in line 6. The discussion continues following the listing.

Listing 11.36 Use the `in` operator to remove duplicates from a container.

```
1  >>> def unique(dups):
2  ...     no_dups = []
3  ...     for item in dups:
4  ...         if item not in no_dups:
5  ...             no_dups.append(item)
6  ...     return no_dups
7  ...
8  >>> xlist = [1, 2, 2, 2, 3, 5, 2]
9  >>> unique(xlist)
10 [1, 2, 3, 5]
11 >>> unique(["Sue", "Joe", "Jose", "Jorge", "Joe", "Sue"])
12 ['Sue', 'Joe', 'Jose', 'Jorge']
```

In line 8 the `list` `xlist` is defined with several copies of the integer `2`. When this is passed to `unique()` in line 9, the `list` that is returned has no duplicates. In line 11 the `unique()` function is passed a `list` of strings with two duplicate entries. The result in line 12 shows these duplicates have been removed.

11.9 Chapter Summary

Python provides the Boolean literals **`True`** and **`False`**.

When used in a conditional statement, all objects are equivalent to either `True` or `False`. Numeric values of zero, empty containers (for example, empty strings and empty `lists`), `None`, and `False` itself are considered to be `False`. All other objects are considered to be `True`.

The **`bool()`** function returns either `True` or `False` depending on whether its argument is equivalent to `True` or `False`.

The template for an **`if`** statement is

```
if <test_expression>:
    <body>
```

The body is executed only if the object returned by the test expression is equivalent to `True`.

`if` statements may have **`elif`** clauses and an **`else`** clause. The template for a general conditional statement is

```
if <test_expression1>:
    <body1>
elif <test_expression2>:
    <body2>
... # Arbitrary number
... # of elif clauses.
else:
    <bodyN>
```

The body associated with the first test expression to return an object equivalent to `True` is executed. No other body is executed. If none of the test expressions returns an object equivalent to `True`, the body associated with the `else` clause, when present, is executed. The `else` clause is optional.

The comparison, or relational, operators compare the values of two operands and return `True` if the implied relationship is true. The comparison operators are: less than (**`<`**), less than or equal to (**`<=`**), greater than (**`>`**), greater than or equal to (**`>=`**), equal to (**`==`**), and not equal to (**`!=`**).

The logical operators **`and`** and **`or`** take two operands. `and` produces a `True` value only if both its operands are equivalent to `True`. `or` produces a `True` value if either or both its operands are equivalent to `True`. The logical operator **`not`** is a unary operator that negates the value of its operand.

All comparison operators have higher precedence than logical operators. `not` has higher precedence than `and`, and `and` has higher precedence than `or`. Parentheses can be used to change the order of precedence of these operators. All math operators have higher precedence than both comparison and logical operators.

`and` and `or` use "shortcircuit" behavior. In expressions involving `and` or `or`, Python only evaluates as much as needed to determine the final outcome. The return value is the object that determines the outcome.

The template for a **`while`**-loop is:

```
while <test_expression>:
    <body>
```

The test expression is checked. If it is equivalent to `True`, the body is executed. The test expression is checked again and the process is repeated.

A **`break`** statement terminates the current loop.

A **`continue`** statement causes the remainder of a loop's body to be skipped in the current iteration of the loop.

Both `break` and `continue` statements can be used with either `for`-loops or `while`-loops.

The **`in`** operator returns `True` if the left operand is contained in the right operand and returns `False` otherwise.

11.10 Review Questions

1. Consider the following code. When prompted for input, the user enters the string `SATURDAY`. What is the output?

```
day = input("What day is it? ")
day = day.lower()
```

```
if day == 'saturday' or day == 'sunday':
   print("Play!")
else:
   print("Work.")
```

2. Consider the following code. When prompted for input, the user enters the string `monday`. What is the output?

```
day = input("What day is it? ")
day = day.lower()
if day != 'saturday' and day != 'sunday':
   print("Yep.")
else:
   print("Nope.")
```

3. Consider the following code. What is the output?

```
values = [-3, 4, 7, 10, 2, 6, 15, -300]
wanted = []
for value in values:
    if value > 3 and value < 10:
        wanted.append(value)

print(wanted)
```

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

```
a = 5
b = 10
if a < b or a < 0 and b < 0:
  print("Yes, it's true.")
else:
  print("No, it's false.")
```

5. What is the value of `x` after the following code executes?

```
x = 2 * 4 - 8 == 0
```

   (a) `True`
   (b) `False`
   (c) None of the above.
   (d) This code produces an error.

6. What is the output generated by the following code?

```
a = 5
b = -10
if a < b or a < 0 and b < 0:
  print("Yes, it's true.")
else:
  print("No, it's false.")
```

7. What is the output generated by the following code?

```
a = -5
b = -10
if (a < b or a < 0) and b < 0:
  print("Yes, it's true.")
else:
  print("No, it's false.")
```

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

```
a = [1, 'hi', False, '', -1, [], 0]
for element in a:
   if element:
      print('T', end=" ")
   else:
      print('F', end=" ")
```

9. Consider the following conditional expression:

```
x > 10 and x < 30
```

Which of the following is equivalent to this?

(a) `x > 10 and < 30`

(b) `10 < x and 30 > x`

(c) `10 > x and x > 30`

(d) `x <= 10 or x >= 30`

10. To what value is `c` set by the following code?

```
a = -3
b = 5
c = a <= (b - 8)
```

(a) `True`

(b) `False`

(c) This code produces an error.

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

```
def is_lower(ch):
   return 'a' <= ch and ch <= 'z'

print(is_lower("t"))
```

(a) `True`

(b) `False`

(c) `None`

(d) This code produces an error

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

```
def is_there(names, query):
   for name in names:
      if query == name:
         return True

print(is_there(['Jake', 'Jane', 'Alice'], 'Tom'))
```

(a) `True`

(b) `False`

(c) `None`

(d) This code produces an error.

13. What output is produced by the following code?

```
def monotonic(xlist):
    for i in range(len(xlist) - 1):
        if xlist[i] < xlist[i + 1]:
            return False
    return True

data1 = [5, 3, 2, 2, 0]
data2 = [5, 2, 3, 2, 0]
print(monotonic(data1), monotonic(data2))
```

(a) `True True`

(b) `True False`

(c) `False True`

(d) `False False`

(e) None of the above.

14. What output is produced by the following code?

```
def swapper(xlist):
    for i in range(len(xlist) - 1):
        if xlist[i] > xlist[i + 1]:
            # Swap values.
            xlist[i], xlist[i + 1] = xlist[i + 1], xlist[i]

data = [5, 3, 2, 2, 0]
swapper(data)
print(data)
```

15. What is the value of `x` after the following code executes?

```
y = 10
x = 2 * 4 - 8 or y
```

(a) `True`

(b) `False`

(c) None of the above.

(d) This code produces an error.

16. What is the value of `x` after the following code executes?

```
y = 10
if 2 * 4 - 8:
    x = 2 * 4 - 8
else:
    x = y
```

(a) `True`

(b) `False`

(c) `0`

(d) `10`

17. What is the value of `x` after the following code executes?

```
x = 4
while x > 0:
    print(x)
    x = x - 1
```

(a) `4`

(b) `1`

(c) `0`

(d) `-1`

(e) None of the above.

18. What is the value of `x` after the following code executes?

```
x = 4
while x == 0:
    print(x)
    x = x - 1
```

(a) `4`

(b) `1`

(c) `0`

(d) `-1`

(e) None of the above.

19. What is the value returned by the function `func1()` when it is called in the following code?

```
def func1(xlist):
    for x in xlist:
        if x < 0:
            return False
    return True

func1([5, 2, -7, 7])
```

(a) `True`

(b) `False`

(c) None of the above.

(d) This code produces an error.

20. What is the value returned by the function `func2()` when it is called in the following code?

```
def func2(xlist):
    for i in range(len(xlist) - 1):
        if xlist[i] + xlist[i + 1] == 0:
            return True
    return False

func2([5, 2, -7, 7])
```

(a) `True`

(b) `False`

(c) None of the above.

(d) This code produces an error.

ANSWERS: 1) `Play!`; 2) `Yep.`; 3) `[4, 7, 6]`; 4) `Yes, it's true.`; 5) a; 6) `No, it's false.`; 7) `Yes, it's true.`; 8) `T T F F T F F`; 9) b; 10) a; 11) a; 12) c; 13) b; 14) `[3, 2, 2, 0, 5]`; 15) c (it is `10`); 16) d; 17) c; 18) a; 19) b; 20) a.