
CHAPTER 3

Operators and String Formatting

Terms in This Chapter

- | | | |
|------------------------|----------------------------------|---|
| ◇ <i>Boolean value</i> | ◇ <i>Format directives</i> | ◇ <i>Operator (%,
Arithmetic, Bitwise,
Comparison,
Conditional, Logical,
Sequence, Shift)</i> |
| ◇ <i>Class</i> | ◇ <i>Hexdump</i> | ◇ <i>String</i> |
| ◇ <i>Concatenation</i> | ◇ <i>Key</i> | ◇ <i>Tuple</i> |
| ◇ <i>Conversion</i> | ◇ <i>Keyword</i> | ◇ <i>Variable</i> |
| ◇ <i>Dictionary</i> | ◇ <i>Literal</i> | |
| ◇ <i>Directive</i> | ◇ <i>Modulus</i> | |
| ◇ <i>Field</i> | ◇ <i>Operator
precedence</i> | |
| ◇ <i>Flag</i> | | |

In this chapter, we'll cover operators and string formatting. Python string formatting controls the creation of strings. Done correctly, it makes the production of these strings simple and straightforward.

I've said it before, and I'll say it again: If you're a beginning programmer, remember that the only way to learn programming is by programming, so try to follow along with the interactive sessions throughout the chapter. The interactive interpreter mode will give you a hands-on understanding of Python operators and string formatting. If you have trouble with an Advanced Topic section, just skim over it; don't let it slow you down.

As in Chapter 2, most of the concepts in this chapter act as building blocks for more complex ideas. Don't worry if something seems unclear to you at this point; you might understand it later, in a different context. For example, logical and comparison operators may not be easily grasped here, but wait until Chapter 4,

44 Chapter 3 Operators and String Formatting

where we deal with the `if` statement, which makes frequent use of these operators and so should clear things up.

If you've programmed before, most of this chapter will be familiar. For example, operators and string formatting in Python and C are very similar. If you have in-depth programming experience, you can probably just skim this material, especially if you're comfortable with C, Java, and/or Visual Basic. Do, however, pay attention to the following sections:

- "Arithmetic with Strings, Lists, and Tuples"
- "% Tuple String Formatting"
- "Advanced Topic: Using the %d, %i, %f, and %e Directives for Formatting Numbers"

Also read the "For Programmers" sidebar (see pages 50–51).

Operators

Recall from Chapter 2 our definition of expressions as simple statements that return a value. In Python, many expressions use operators, such as `+`, `-`, `*`, and `=`. The following subsections describe each operator type, and each section contains a table of the type's operators along with sample interactive sessions illustrating their use. If you feel as if you've been this way before, you have—we've been using operators since Chapter 1.

Arithmetic Operators

Arithmetic operators work with the numeric types `Float`, `Int`, and `Long`. Table 3–1 describes them, including three we have yet to encounter: modulus (`%`), which gives the remainder; exponential (`**`), which raises one number to the power of another number; and `abs`, which gives a number's absolute value.

One example of modulus is $3/2$, which gives the remainder of 1 ($3/2 = 1\frac{1}{2}$). Another is $10/7$, which gives a remainder of 3 ($10/7 = 1\frac{3}{7}$). In Python, we express the previous sentence as

```
>>> 10 % 7
3
>>> 3 % 2
1
>>>
```

Once you understand modulus, the `divmod()` function, which we'll discuss in a later chapter, should come easily to you.

Table 3–1 *Arithmetic Operators*

Operator	Description	Interactive Session
+	Addition	<pre>>>> x = 1 + 2 >>> print (x) 3</pre>
-	Subtraction	<pre>>>> x = 2 - 1 >>> print (x) 1</pre>
*	Multiplication	<pre>>>> x = 2 * 2 >>> print (x) 4</pre>
/	Integer division returns an Integer type; float division returns a float type	<p>Integer division:</p> <pre>>>> x = 10 / 3 >>> print (x) 3</pre> <p>Float division:</p> <pre>>>> x = 10.0 / 3.3333 >>> print (x) 3.000030000300003</pre>
%	Modulus—gives the remainder; typically used for integers	<pre>>>> x = 10 % 3 >>> print (x) 1</pre>
**	Exponential	<pre>>>> x = 10**2 >>> print(x) 100</pre>
divmod	Does both of the division operators at once and returns a tuple; the second item in the tuple contains the remainder. <code>divmod(x,y)</code> is equivalent to <code>x/y,x%y</code>	<p>This:</p> <pre>>>> divmod (10,3) (3, 1)</pre> <p>Is the same as this:</p> <pre>>>> 10/3,10%3 (3, 1) <p>This:</p> <pre>>>> divmod (5,2) (2, 1)</pre> <p>Is the same as this:</p> <pre>>>> 5/2, 5%2 (2, 1)</pre> </pre>
abs	Finds the absolute value of a number	<pre>>>> abs(100) 100 >>> abs(-100) 100</pre>
-, +	Sign	<pre>>>> 1, -1, +1, +-1 (1, -1, 1, -1)</pre>

Numeric Conversion Operators

Many times we need to convert from one numeric type to another. The three operators that perform this conversion are `Int(x)`, `Long(x)`, and `Float(x)`, where `x` is any numeric value. To illustrate, in the example that follows we create three numeric types: `l` (`Long`), `f` (`Float`), and `i` (`Integer`).

```
>>> l, f, i = 1L, 1.0, 1
```

The output is

```
>>> l, f, i
(1L, 1.0, 1)
```

The next three examples in turn convert `i` to `Float`, `f` and `i` to `Long`, and `l` and `f` to `Integer`.

```
>>> float(i)
1.0
>>> float(l)
1.0
>>> long(f), long(i)
(1L, 1L)

>>> int(l), int(f)
(1, 1)
>>>
```

Logical Operators, Comparison Operators, and Boolean Values

Logical operators are a way to express choices, such as “This one *and* that one *or* that one but *not* this one.” Comparison operators are a way to express questions, such as “Is this one *greater than* that one?” Both work with Boolean values, which express the answer as either true or false. Unlike Java, Python has no true `Boolean` type. Instead, as in C, its Booleans can be numeric values, where any nonzero value must be true and any zero value must be false. Thus, Python interprets as false the following values:

- None
- Empty strings
- Empty tuples
- Empty lists
- Empty dictionaries
- Zero

and as true all other values, including

- Nonempty strings
- Nonempty tuples
- Nonempty lists

- Nonempty dictionaries
- Not zero

Table 3–2 describes the logical operators. They return 1 for a true expression and 0 for a false expression. Table 3–3 describes the comparison operators. They return some form of true for a true expression and some form of false for a false expression.

Logical and comparison operators often work together to define application logic (in English, application logic simply means decision making). When they do, they're often used with `if` and `while` statements. Don't worry about `if` and `while` just yet; we'll get into them in detail in Chapter 4. For now, a simple way to visualize them is to imagine that you like vanilla and chocolate ice cream but hate nuts, and you want to express your preference in a way that Python will understand, like this:

```
if (flavor == chocolate or flavor == vanilla and \
    not nuts and mycash > 5):
    print("yummy ice cream give me some")

while(no_vanilla_left and no_chocolate_left ):
    print ("no more ice cream for me")
```

Table 3–2 *Logical Operators*

Operator	Description	Interactive Session
<code>and</code>	And two values or comparisons together	<pre>>>> x,y = 1,0 >>> x and y 0 >>> x,y = 1,1 >>> x and y 1</pre>
<code>or</code>	Or two values together	<pre>>>> x,y = 1,0 >>> x or y 1 >>> x,y = 0,0 >>> x or y 0</pre>
<code>not</code>	Inverse a value	<pre>>>> x,y = 0,0 >>> not x 1 >>> not y 1 >>> x = 1 >>> not x 0</pre>

48 Chapter 3 Operators and String Formatting

Table 3-3 Comparison Operators

Operator	Description	Interactive Session
==	Equal to	<pre>>>> x,y,z=1,1,2 >>> x==y 1 >>> x==z 0</pre>
>=	Greater than or equal to	<pre>>>> z>=x 1 >>> x>=z 0</pre>
<=	Less than or equal to	<pre>>>> z<=x 0 >>> x<=z 1</pre>
>	Greater than	<pre>>>> x>z 0 >>> z>x 1</pre>
<	Less than	<pre>>>> x<z 1 >>> z<x 0</pre>
!=	Not equal to	<pre>>>> x!=y 0 >>> x!=z 1</pre>
<>	Not equal to	<pre>>>> x<>y 0 >>> x<>z 1</pre>
is	Object identity	<pre>>>> str = str2 = "hello" >>> str is str2 1 >>> str = "hi" >>> str is str2 0</pre>
is not	Negated object identity	<pre>>>> s1 = s2 = "hello" >>> s1 is not s2 0 >>> s1 = s2 + " Bob" >>> s1 is not s2 1</pre>

Table 3-3 *Continued*

Operator	Description	Interactive Session
in	Checks to see if an item is in a sequence	<pre>>>> list = [1,2,3] >>> 1 in list 1 >>> 4 in list 0</pre>
not in	Checks to see if an item is NOT in a sequence	<pre>>>> list = [1,2,3] >>> 1 not in list 0 >>> 4 not in list 1</pre>

Advanced Topic: Logical Operators and Boolean Returns

Comparison operators always return either 1 or 0 of type Integer.

```
>>> 1 > 2
0
>>> 1 < 2
1
```

Logical operators can return more than the Integer types 1 or 0, as we see in the following expression, which determines if 0 or (1,2,3) is true.

```
>>> 0 or (1,2,3)
(1, 2, 3)
```

Python equates 0 to false and a nonempty tuple (1, 2, 3) to true, so the logical operator returns the true statement, that is, the (1,2,3) tuple literal.

The following expression determines if 1 < 2 or the integer 5 is true:

```
>>> 1 < 2 or 5
1
```

Because 1 < 2 is true, the expression returns 1, which equates to true, but it equates 5 to true as well; however, only the first true statement in an or statement (1 above) is returned.

The next expression also determines if 1 < 2 or the integer 5 is true, but this time we swap the operands.

```
>>> 5 or 1 < 2
5
```

50 Chapter 3 Operators and String Formatting

Once again, only the first true statement is returned, which is now 5.

Like `or`, `and` returns the first true operand. However, `and` is unlike `or` in that only the last operand can make it true.

```
>>> (1,1) and [2,2]
[2, 2]
>>> [2,2] and (1,1)
(1, 1)
>>> [2,2] and (3,3) and {"four":4}
{'four': 4}
```

Conversely, the first occurrences of a false are returned by `and`.

```
>>> [1,1] and {} and ()
{}
>>> (1,1) and [] and {}
[]
```

For Programmers: Conditional Operators in Other Languages

C, C++, and Java have a conditional operator that works conveniently as shorthand for `and` and `or`. In Java, for example, the following two `if` statements are equivalent:

```
val = boolean_test ? true_return : false_return;

if (boolean_test)
    val = true_return;
else
    val = false_return;
```

Python has no conditional operator, but you can simulate one with the form

```
val = (boolean_test and true_return) \
    or false_return
```

This works because `or` always returns the first true statement and `and` always returns the last one, so these two statements are equivalent:

```
>>> if ( 3 > 5):
...     num = 1
... else:
...     num = 2
...
>>> num
2

>>> num = (3>5 and 1) or 2
>>> num
2
```

The following two expressions are also equivalent:

```
>>> if ( 5 > 3):
...     num = 1
```

```
... else:
...     num = 2
...
>>> num
1

>>> num = (5>3 and 1) or 2
>>> num
1
```

Be warned: This simulation works only if the expressions `true_return` and `false_return` are *not* equivalent to false. If we need them to be false, we can do something like this:

```
>>> true_return = 0
>>> false_return = 2
>>> num = (5 > 3 and [true_return]) \
         or [false_return]
>>> num
[0]
```

Now the `num` variable is equivalent to a list containing one element, but this isn't exactly what we want. However, since this code is returning a list, we can put the entire expression to the left of the assignment operator in parentheses, which will achieve our desired result.

```
>>> num = ((5 > 3 and [true_return]) or [false_return])[0]
>>> num
0
```

To sum up, Python's bulletproof equivalent of the conditional operator is

```
val = ((boolean_test and [true_return]) \
      or [false_return])[0]
```

which is no more verbose than another Python expression:

```
if (boolean_test): val = true_return
else: val = false_return
```

Python's version of the conditional operator is hard to understand and use, so go easy with it. Perhaps one day Python will have a conditional operator of its own (and, I might add, its own `+=` operator).

Advanced Topic: Bitwise and Shift Operators

If you lack experience with any programming language or with Boolean algebra, you should ignore bitwise operators. Another reason to ignore them is that they're usually associated with low-level programming, and you're learning Python, which is much higher level than C, C++, or even Java. If for some reason you're curious about bitwise operators, any introductory C text will tell you all you need to know. The same goes for shift operators.

52 Chapter 3 Operators and String Formatting

Just for the sake of completeness, Table 3–4 describes both operator types. To understand it, you need to know something of hexadecimal and Boolean algebra. (See Chapter 10 for an example of a hexdump file viewer, which uses the shift operators.)

Operator Precedence

Operator precedence determines the order in which Python evaluates the parts of a statement. It generally follows the operator precedence you learned in high school algebra and is nearly identical to that used in any other common programming language. Here's an example in which y/z is processed before $2 + y$, rendering x equal to 4 and not 6.

```
>>> x,y,z=1,4,2
>>> x = 2 + y / z
>>> x
4
```

When in doubt as to which operator will be evaluated first, use parentheses. They prevent many a mistake if you use them to force precedence, and they enhance code readability.

You may occasionally want to force a precedence other than the algebraic default to make it more explicit. The following example shows how to do this:

```
>>> x,y,z = 1,4,2
>>> x = 2 + (y/z)
>>> x
4

>>> x = (2+y) /z
>>> x
3
```

The choice of precedence here depends on which expression— $2 + y$ or y/z —is to be evaluated first. Note, though, that the value of x changes according to the grouping and ends up as either 4 or 3. The first expression, $2 + y/z$, is unnecessary except to adorn the code with parentheses for clarity, which is important for code maintainability.

Visit www.python.org for a detailed description of operator precedence. For now, the following list shows all operators in their precedence order:

- | | |
|--------------------------------|------------------------------------|
| • <code>[], (), {}, ''</code> | Parentheses, string conversion |
| • <code>seq[index]</code> | Indexing sequences or dictionaries |
| • <code>integer.MAX_INT</code> | Attribute reference |
| • <code>~I</code> | Bit inversion |
| • <code>-i, +i</code> | Unary minus, unary plus |
| • <code>*, /, %</code> | Multiplication, division, modulus |
| • <code>+, -</code> | Addition, subtraction |

Table 3-4 *Bitwise and Shift Operators*

Operator	Description	Interactive Session
<<	Shift left	<pre>>>> # binary 1111 1111 >>> x = 0xff >>> # z = 0011 1111 1100 >>> z = x << 2 >>> print (x) 255 >>> print (z) 1020</pre>
>>	Shift right	<pre>>>> # z = 1020 >>> #z = 1111 1111 >>> z = z >> 2 >>> z 255</pre>
&	Bitwise and	<pre>>>> # y = binary 0000 1010 >>> y = 0x0A >>> print (y) 10 >>> print (x) 255 >>> z = y & z >>> print (z) 10</pre>
	Bitwise or	<pre>>>> #continued example >>> z = y x >>> print (z) 255</pre>
^	Bitwise XOR	<pre>>>> z = y ^ x >>> z 245</pre>
~	Bitwise not	<pre>>>> y = 0xffffffff >>> y -1 >>> z = ~y >>> z 0</pre>

Table 3-5 *Sequence Operators*

Operator	Description	Interactive Session
[i index]	Get the indexed item	<pre>>>> nums = (0,1,2,3,4,5,6,7,8,9) >>> nums[0] 0 >>> nums[1] 1</pre>
[:]	Slice notation	<pre>>>> nums = (0,1,2,3,4,5,6,7,8,9) >>> nums [:] (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) >>> nums [5:] (5, 6, 7, 8, 9) >>> nums [:5] (0, 1, 2, 3, 4)</pre>
len()	Length	<pre>>>> nums = (0,1,2,3,4,5,6,7,8,9) >>> len (nums) 10</pre>
max()	Get the largest item in the sequence	<pre>>>> nums = (0,1,2,3,4,5,6,7,8,9) >>> max(nums) 9 >>> letters = "abcdefg" >>> max (letters) 'g'</pre>
min()	Get the smallest item in the sequence	<pre>>>> nums = (0,1,2,3,4,5,6,7,8,9) >>> min(nums) 0 >>> letters = "abcdefg" >>> min (letters) 'a'</pre>

Formatting Strings—Modulus

Although not actually modulus, the Python % operator works similarly in string formatting to interpolate variables into a formatting string. If you've programmed in C, you'll notice that % is much like C's printf(), sprintf(), and fprintf() functions.

56 Chapter 3 Operators and String Formatting

There are two forms of %, one of which works with strings and tuples, the other with dictionaries.

```
StringOperand % TupleOperand
```

```
StringOperand % DictionaryOperand
```

Both return a new formatted string quickly and easily.

% Tuple String Formatting

In the `StringOperand % TupleOperand` form, `StringOperand` represents special directives within the string that help format the tuple. One such directive is `%s`, which sets up the format string

```
>>> format = "%s is my friend and %s is %s years old"
```

and creates two tuples, `Ross_Info` and `Rachael_Info`.

```
>>> Ross_Info = ("Ross", "he", 28)
>>> Rachael_Info = ("Rachael", "she", 28)
```

The format string operator (%) can be used within a `print` statement, where you can see that every occurrence of `%s` is respectively replaced by the items in the tuple.

```
>>> print (format % Ross_Info)
Ross is my friend and he is 28 years old

>>> print (format % Rachael_Info)
Rachael is my friend and she is 28 years old
```

Also note that `%s` automatically converts the last item in the tuple to a reasonable string representation. Here's an example of how it does this using a list:

```
>>> bowling_scores = [190, 135, 110, 95, 195]
>>> name = "Ross"
>>> strScores = "%s's bowling scores were %s" \
...           % (name, bowling_scores)
>>> print strScores
Ross's bowling scores were [190, 135, 110, 95, 195]
```

First, we create a list variable called `bowling_scores` and then a string variable called `name`. We then use a string literal for a format string (`StringOperand`) and use a tuple containing `name` and `bowling_scores`.

Format Directives

Table 3-6 covers all of the format directives and provides a short example of usage for each. Note that the tuple argument containing a single item can be denoted with the % operator as `item`, or `(item)`.

Table 3-6 *Format Directives*

Directive	Description	Interactive Session
%s	Represents a value as a string	<pre>>>> list = ["hi", 1, 1.0, 1L] >>> "%s" % list "['hi', 1, 1.0, 1L]" >>> "list equals %s" % list "list equals ['hi', 1, 1.0, 1L]"</pre>
%i	Integer	<pre>>>> "i = %i" % (5) 'i = 5' >>> "i = %3i" % (5) 'i = 5'</pre>
%d	Decimal integer	<pre>>>> "d = %d" % 5 'd = 5' >>> "%3d" % (3) ' 3'</pre>
%x	Hexadecimal integer	<pre>>>> "%x" % (0xff) 'ff' >>> "%x" % (255) 'ff'</pre>
%X	Hexadecimal integer	<pre>>>> "%X" % (0xff) 'FF' >>> "%X" % (255) 'FF'</pre>
%o	Octal integer	<pre>>>> "%o" % (255) 377 >>> "%o" % (0377) 377</pre>
%u	Unsigned integer	<pre>>>> print "%u" % -2000 2147481648 >>> print "%u" % 2000 2000</pre>
%e	Float exponent	<pre>>>> print "%e" % (30000000L) 3.000000e+007 >>> "%5.2e" % (30000000L) '3.00e+008'</pre>

(continued)

58 Chapter 3 Operators and String Formatting

Table 3-6 *Format Directives (continued)*

Directive	Description	Interactive Session
%f	Float	<pre>>>> "check = %1.2f" % (3000) 'check = 3000.00' >>> "payment = \$%1.2f" % 3000 'payment = \$3000.00'</pre>
%g	Float exponent	<pre>>>> "%3.3g" % 100 '100.' >>> "%3.3g" % 1000000000000L '10.e11' >>> "%g" % 100 '100.'</pre>
%c	ASCII character	<pre>>>> "%c" % (97) 'a' >>> "%c" % 97 'a' >>> "%c" % (97) 'a'</pre>

Table 3-7 shows how flags can be used with the format directives to add leading zeroes or spaces to a formatted number. They should be inserted immediately after the %.

Table 3-7 *Format Directive Flags*

Flag	Description	Interactive Session
#	Forces octal to have a 0 prefix; forces hex to have a 0x prefix	<pre>>>> "%#x" % 0xff '0xff' >>> "%#o" % 0377 '0377'</pre>
+	Forces a positive number to have a sign	<pre>>>> "%+d" % 100 '+100'</pre>
-	Left justification (default is right)	<pre>>>> "%-5d, %-5d" % (10, 10) '10 , 10 '</pre>
" "	Precedes a positive number with a blank space	<pre>>>> "% d,% d" % (-10, 10) '-100,10'</pre>
0	0 padding instead of spaces	<pre>>>> "%05d" % (100,) '00100'</pre>

Advanced Topic: Using the %d, %i, %f, and %e Directives for Formatting Numbers

The % directives format numeric types: %i works with Integer; %f and %e work with Float with and without scientific notation, respectively.

```
>>> "%i, %f, %e" % (1000, 1000, 1000)
'1000, 1000.000000, 10.000000e+002'
```

Notice how awkward all of those zeroes look. You can limit the length of precision and neaten up your code like this:

```
>>> "%i, %2.2f, %2.2e" % (1000, 1000, 1000)
'1000, 1000.00, 10.00e+002'
```

The %2.2f directive tells Python to format the number as at least two characters and to cut the precision to two characters after the decimal point. This is useful for printing floating-point numbers that represent currency.

```
>>> "Your monthly payments are $%1.2f" % (payment)
'Your monthly payments are $444.43'
```

All % directives have the form %min.precision(type), where min is the minimum length of the field, precision is the length of the mantissa (the numbers on the right side of the decimal point), and type is the type of directive (e, f, i, or d). If the precision field is missing, the directive can take the form %min(type), so, for example, %5d ensures that a decimal number has at least 5 fields and %20f ensures that a floating-point number has at least 20.

Let's look at the use of these directives in an interactive session.

```
>>> "%5d" % (100,)
'   100'
>>> "%20f" % (100,)
'          100.000000'
```

Here's how to truncate the float's mantissa to 2 with %20.2f.

```
>>> "%20.2f" % (100,)
'          100.00'
```

The padding that precedes the directive is useful for printing rows and columns of data for reporting because it makes the printed output easy to read. This can be seen in the following example (from *format.py*):

```
# Create two rows
row1 = (100, 10000, 20000, 50000, 6000, 6, 5)
row2 = (1.0, 2L, 5, 2000, 56, 6.0, 7)

#
# Print out the rows without formatting
print "here is an example of the columns not lining up"
print `row1` + "\n" + `row2`
print
```

60 Chapter 3 Operators and String Formatting

```

#
# Create a format string that forces the number
# to be at least 3 characters long to the left
# and 2 characters to the right of the decimal point
format = "(%3.2e, %3.2e, %3.2e, %3.2e, " + \
          "%3.2e, %3.2e, %3.2e)"

#
# Create a string for both rows
# using the format operator
strRow1 = format % row1
strRow2 = format % row2
print "here is an example of the columns" + \
      " lining up using %e"

print strRow1 + "\n" + strRow2
print

# Do it again this time with the %i and %d directive
format1 = "(%6i, %6i, %6i, %6i, %6i, %6i, %6i)"
format2 = "(%6d, %6d, %6d, %6d, %6d, %6d, %6d)"
strRow1 = format1 % row1
strRow2 = format2 % row2
print "here is an example of the columns" + \
      " lining up using %i and %d"

print strRow1 + "\n" + strRow2
print

here is an example of the columns not lining up
(100, 10000, 20000, 50000, 6000, 6, 5)
(1.0, 2L, 5, 2000, 56, 6.0, 7)

here is an example of the columns lining up using %e
(1.00e+002, 1.00e+004, 2.00e+004, 5.00e+004, 6.00e+003, 6.00e+000, 5.00e+000)
(1.00e+000, 2.00e+000, 5.00e+000, 2.00e+003, 5.60e+001, 6.00e+000, 7.00e+000)

here is an example of the columns lining up using %i and %d
( 100,    10000,    20000,    50000,    6000,    6,    5)
(   1,     2,      5,      2000,     56,     6,    7)

```

You can see that the `%3.2e` directive permits a number to take up only three spaces plus the exponential whereas `%6d` and `%6i` permit at least six spaces. Note that `%i` and `%d` do the same thing that `%e` does. Most C programmers are familiar with `%d` but may not be familiar with `%i`, which is a recent addition to that language.

String % Dictionary

Another useful Python feature for formatting strings is `StringOperand % DictionaryOperand`. This form allows you to customize and print named fields in the string. `%(Income)d` formats the value referenced by the `Income` key. Say, for example, that you have a dictionary like the one here:

```
Monica = {
    "Occupation": "Chef",
    "Name" : "Monica",
    "Dating" : "Chandler",
    "Income" : 40000
}
```

With `%(Income)d`, this is expressed as

```
>>> "%(Income)d" % Monica
'40000'
```

Now let's say you have three best friends, whom you define as dictionaries named Monica, Chandler, and Ross.

```
Monica = {
    "Occupation": "Chef",
    "Name" : "Monica",
    "Dating" : "Chandler",
    "Income" : 40000
}

Ross = {
    "Occupation": "Scientist Museum Dude",
    "Name" : "Ross",
    "Dating" : "Rachael",
    "Income" : 70000
}

Chandler = {
    "Occupation": "Buyer",
    "Name" : "Chandler",
    "Dating" : "Monica",
    "Income" : 65000
}
```

To write them a form letter, you can create a format string called `message` that uses all of the above dictionaries' keywords.

```
message = "%(Name)s, %(Occupation)s, %(Dating)s," \
          "%(Income)2.2f"
```

Notice that `%(Income)2.2f` formats this with a floating-point precision of 2, which is good for currency. The output is

```
Chandler, Buyer, Monica, 65000.00
Ross, Scientist Museum Dude, Rachael, 70000.00
Monica, Chef, Chandler, 40000.00
```

You can then print each dictionary using the format string operator.

```
print message % Chandler
print message % Ross
print message % Monica
```

62 Chapter 3 Operators and String Formatting

To generate your form letter and print it out to the screen, you first create a format string called `dialog`.

```
dialog = """
Hi %(Name)s,

How are you doing? How is %(Dating)s?
Are you still seeing %(Dating)s?

How is work at the office?
I bet it is hard being a %(Occupation)s.
I know I could not do it.
"""
```

Then you print out each dictionary using the `dialog` format string with the `%` format string operator.

```
print dialog % Ross
print dialog % Chandler
print dialog % Monica
```

The output is

```
Hi Ross,

How are you doing? How is Rachael?
Are you still seeing Rachael?

How is work at the office?
I bet it is hard being a Scientist Museum Dude.
I know I could not do it.

Hi Chandler,
How are you doing? How is Monica?
Are you still seeing Monica?

How is work at the office?
I bet it is hard being a Buyer.
I know I could not do it.

Hi Monica,
How are you doing? How is Chandler?
Are you still seeing Chandler?

How is work at the office?
I bet it is hard being a Chef.
I know I could not do it.
```

`%(Income)d` is a useful, flexible feature. You just saw how much time it can save you in writing form letters. Imagine what it can do for writing reports.

Summary

String formatting is the way we organize instructions so that Python can understand how to incorporate data in the creation of strings. How strings are formatted determines the presentation of this data. The basis of string formatting is its use of the formatting directives.

Logical operators (`and`, `or`, `not`) return true or false values. Comparison operators (`in`, `not in`, `is`, `is not`, `==`, `!=`, `<>`, `>`, `>=`, `<`, `<=`) compare two values—is one value greater than another, or is one value not equal to another? Comparisons always return 1 for true and 0 for false.

Any value in Python equates to a Boolean true or false. A false can be equal to none, a zero of numeric type (`0`, `01`, `0.0`), an empty sequence (`'`, `()`, `[]`), or an empty dictionary (`{}`). All other values are considered true.

Sequences, tuples, lists, and strings can be added or multiplied by a numeric type with the addition and multiplication operators (`+`, `*`), respectively. Strings can be formatted with tuples and dictionaries using the format directives `%i`, `%d`, `%f`, and `%e`. Formatting flags can be used with these directives.