Chapter 2

Core Basics

In this chapter we introduce some of the basic concepts that are important for understanding what is to follow. We also introduce a few of the fundamental operations that are involved in the vast majority of useful programs.

2.1 Literals and Types

Section 1.3 introduced the string literal. A string literal is a collection of characters enclosed in quotes. In general, a *literal* is code that exactly represents a value. In addition to a string literal, there are numeric literals.

There are actually a few different types of numeric values in Python. There are integer values which correspond to whole numbers (i.e., the countable numbers that lack a fractional part). In Python, an integer is known as an `int`. A number that can have a fractional part is called a `float` (a `float` corresponds to a real number). For example, `42` is an `int` while `3.14` is a `float`. The presence of the decimal point makes a number a `float`—it doesn't matter if the fractional part is zero. For example, the literal `3.0`, or even just `3.`, are also `float`s[1] despite the fact that these numbers have fractional parts of zero.[2]

There are some important differences between how computers work with `int`s and `float`s. For example, `int`s are stored exactly in the computer while, in general, `float`s are only stored approximately. This is a consequence of the fact that we are entering numbers as decimal numbers, i.e., numbers in the base 10 counting system, but the computer stores these numbers internally as a finite number of binary (base 2) digits. For integers we can represent a number exactly in any counting system. However, for numbers with fractional parts, this is no longer true. Consider the number one-third, i.e., 1/3. To represent this as a decimal number (base 10) requires an infinite number of digits: $0.3333\cdots$. But, if one uses the base 3 (ternary) counting system, one-third is

From the file: `core-basics.tex`

[1]We will write the plural of certain nouns that have meaning in Python as a mix of fonts. The noun itself will be in `Courier` and the trailing "s" will be in Times-Roman, e.g., `float`s.

[2]Other numeric data types that Python provides are `complex`, `Decimal`, and `Fraction`. Of these, only `complex` numbers can be entered directly as literals. They will be considered in Chap. 8. `Decimal`s provide a way of representing decimal numbers in a manner that is consist with how humans think of decimal numbers and do not suffer from the approximation inherent in `float`s. `Fraction`s are rational numbers where the numerator and denominator are integers.

represented with a single digit to the right of the decimal sign, i.e., 0.1_3.[3] This illustrates that when converting from one counting system to another (i.e., one base to another), the fractional part of the number that could be represented by a finite number of digits in one counting system may require an infinite number of digits in the other counting system. Because computers cannot store an infinite number of digits, the finite number of digits that are stored for a `float` is an approximation of the true number.[4]

As mentioned in Sec. 1.6.1, when we enter a literal in the interactive environment, Python merely echoes the literal back to us. This is true of both numeric and string literals as Listing 2.1 illustrates.

Listing 2.1 Entry of numeric and string literals in the interactive environment.

```
>>> 7         # int literal.
7
>>> 3.14      # float literal.
3.14
>>> -2.00001 # Negative numbers are indicated with a negative sign.
-2.00001
>>> +2.00     # Can explicitly use positive sign, but not needed.
2.0
>>> 1.23e3    # Exponential notation: positive exponent.
1230.0
>>> 1.23e-3   # Exponential notation: negative exponent.
0.00123
>>> 1e0       # Exponential notation: decimal point not required.
1.0
>>> "Literally unbelievable." # String in double quotes.
'Literally unbelievable.'
>>> 'Foo the Bar'             # String in single quotes.
'Foo the Bar'
```

Line 5 shows the entry of a negative number. One can use the positive sign to indicate a positive number, as is done in line 7, but if no sign is provided, the number is assumed to be positive.

Lines 9, 11, and 13 demonstrate the entry of a number using exponential notation. From your calculator you may already be familiar with the exponential notation in which the letter `e` is used as shorthand for "$\times 10$" and the number to the right of `e` is the exponent of 10. Thus, when we write `1.23e-3` we mean 1.23×10^{-3} or 0.00123. Note think in line 13 we have entered `1e0` which is equivalent to $1 \times 10^0 = 1$. You might think that since there is no decimal point, `1e0` should be an integer. However, the output of `1.0` shown on line 14 indicates that it is actually a `float`. So, when a decimal point appears in a number *or* when `e` appears in a number, the number is a `float`.

In lines 15 and 17 string literals are entered. In line 15 the string is enclosed in double quotes but what Python echoes back in line 16 is enclosed in single quotes. Line 17 shows that we can

[3]The subscript 3 indicates this is a number in base 3. If you are unfamiliar with counting in numbering systems other than 10, it is not a concern. We will revisit this issue later.

[4]For a `float` one obtains about 15 decimal digits of accuracy. Thus, although a `float` may be an approximation of the true number, 15 digits of accuracy is typically sufficient to meet the needs of an application.

also use single quotes to enter a string literal. In line 18 Python echoes this back and again uses single quotes.

The *type* of various things, including literals, can be determined using the `type()` function.[5] In finished programs it is rare that one needs to use this function, but it can be useful for debugging and helping to understand code. Listing 2.2 illustrates what happens when various literals are passed as an argument to `type()`.

Listing 2.2 Using the `type()` function to determine the type of various literals.

```
 1  >>> type(42)
 2  <class 'int'>
 3  >>> type(-27.345e14)
 4  <class 'float'>
 5  >>> type("Hello World!")
 6  <class 'str'>
 7  >>> type("A") # Single character, double quotes.
 8  <class 'str'>
 9  >>> # Single character, single quotes; looks like int but isn't!
10  >>> type('2')
11  <class 'str'>
12  >>> type("")  # Empty string.
13  <class 'str'>
14  >>> type('-27.345e14')  # Looks like float but isn't!
15  <class 'str'>
```

In line 1 the programmer asks for the type of the literal `42`. Line 2 reports that it is an `int`. The response is a little more involved than simply saying this is an `int`—the word `class` indicates that `42` is actually an instance of the `int` class. The notion of an "instance of a class" relates to object oriented programming concepts and is something that doesn't concern us yet. Suffice it simply to say that "`42` is an `int`."

In line 3 a `float` is passed as an argument to `type()` while in line 5 the argument is a string. Line 6 shows that Python identifies a string as a `str`. We will take string to be synonymous with `str` and integer to be synonymous with `int`. (For `float`s we will simply write `float`.)

Lines 7 through 11 show that a single character, regardless of the type of quotation marks that enclose it, is considered a string.[6] Lines 12 and 13 show the type of an *empty string*: there are no characters enclosed between the quotation marks in line 12 and yet this is still considered a string. An empty string may seem a rather useless literal, but we will see later that it plays a useful role in certain applications.

Lines 10 and 14 of Listing 2.2 are of interest in that, to a human, the characters in the string represent a numeric value; *nevertheless*, the literal is, in fact, a string. Keep in mind that just because something might appear to be a numeric value doesn't mean it is. To hint at how this may lead to some confusion, consider the code in Listing 2.3. In line 1 the `print()` function has two

[5]Instead of saying "various things," it would be more correct to say the *objects* in our code. We discuss what objects are in Chap. 5.

[6]This is not the case in some other languages such as C and C++.

arguments: an `int` and a `str`. However, in line 2 the output produced for each of these arguments is identical.

Listing 2.3 Printing numeric values and strings that look like numeric values.

```
1 >>> print(42, "42") # An int and a str that looks like an int.
2 42 42
3 >>> print('3.14')   # A str that looks like a float.
4 3.14
5 >>> print(3.14)     # A float.
6 3.14
```

In line 3 a `str` is printed while in line 5 a `float` is printed. The output of these two statements is identical. But, there are things that we can do with numeric values that we cannot do with strings and, conversely, there are things we can do with strings that we cannot do with numeric values. Thus, we need to be somewhat mindful of type. When bugs are encountered, often the best tool for debugging is the `print()` function. However, as the code in Listing 2.3 demonstrates, if we ever encounter problems related to type, the `print()` function, by itself, might not be the best tool for sorting out the problem since it may display data of different types identically.

2.2 Expressions, Arithmetic Operators, and Precedence

An *expression* is code that returns data, or, said another way, evaluates to a value.[7] In Python, it is more correct to say that an expression returns an *object* instead of a *value* or *data*, but we will ignore this distinction for now (and return to it later). The literal values considered in Sec. 2.1 are the simplest form of an expression. When we write a literal, we are providing (fixed) data. We can also have expressions that evaluate to a new value. For example, we can add two numbers that may be entered as literals. The result of this expression is the new value given by the sum of the numbers. When an expression is entered in an interactive environment, Python will display the result of the expression, i.e., the value to which the expression evaluates.

Python provides the standard arithmetic operations of addition, subtraction, multiplication, and division which are invoked via the *operators* `+`, `-`, `*`, and `/`, respectively. The numbers to either side of these operators are known as the *operands*. Listing 2.4 demonstrates the use of these operators.

Listing 2.4 Demonstration of the basic arithmetic operators.

```
>>> 4 - 5          # Difference of two ints.
-1
>>> 5 + 4          # Sum of two ints.
9
>>> 5 + 4.         # Sum of int and float.
```

[7]Rather than adhering to traditional usage where the noun *data* is only plural (with a singular form of datum), we will adopt modern usage where data can be either singular or plural.

```
 6  9.0
 7  >>> 5 + 4 * 3     # * has higher precedence than +.
 8  17
 9  >>> (5 + 4) * 3   # Parentheses can change order of operations.
10  27
11  >>> 5 / 4         # float division of two ints.
12  1.25
13  >>> 10 / 2        # float division of two ints.
14  5.0
15  >>> 20 / 4 / 2    # Expressions evaluated left to right.
16  2.5
17  >>> (20 / 4) / 2 # Parentheses do not change order of operation here.
18  2.5
19  >>> 20 / (4 / 2) # Parentheses do change order of operation here.
20  10.0
```

Lines 1 through 4 show that the sum or difference of two `int`s is an `int`. Lines 5 and 6 show that the sum of an `int` and a `float` yields a `float` (as indicated by the decimal point in the result in line 6).[8] When an arithmetic operation involves an `int` and a `float`, Python has to choose a data type for the result. Regardless of whether or not the fractional part of the result is zero, the result is a `float`. Specifically, the `int` operand is converted to a `float`, the operation is performed, and the result is a `float`.

The expression in line 7 uses both addition and multiplication. We might wonder if the addition is done first, since it appears first in the expression, or if the multiplication is done first since in mathematics multiplication has higher precedence than addition. The answer is, as indicated by the result on line 8, that Python follows the usual mathematical rules of precedence: Addition and subtraction have lower precedence than multiplication and division. Thus, the expression `5 + 3 * 4` is equivalent to `5 + (3 * 4)` where operations in parentheses always have highest precedence. Line 9 shows that we can use parentheses to change the order of operation.

Lines 11 and 13 show the "`float` division" of two `int`s. In line 11 the numerator and denominator are such that the quotient has a non-zero fractional part and thus we might anticipate obtaining a `float` result. However, in line 13, the denominator divides evenly into the numerator so that the quotient has a fractional part of zero. Despite this fact, the result is a `float`. This is a consequence of the fact that we are using a single slash (`/`) to perform the division. Python also provide another type of division, floor division, which is discussed in Sec. 2.8.

In line 15 there are two successive divisions: `20 / 4 / 2`. Here the operations are evaluated left to right. First `20` is divided by `4` and then the result of that is divided by `2`, ultimately yielding `2.5`. If we are ever unsure of the order of operation (or if we merely think it will enhance the clarity of an expression), we can always add parentheses as line 17 illustrates. Lines 15 and 17 are completely equivalent and the parentheses in line 17 are merely "decorative."

The operators in Listing 2.4 are *binary operators* meaning that they take two operands—one operand is to the left of the operator and the other is to the right. However, plus and minus signs can also be used as *unary operators* where they take a single operand. This was illustrated in lines

[8]Note that the blank spaces that appear in these expressions are merely used for clarity. Thus `5 + 4` is equivalent to `5+4`.

5 and 7 of Listing 2.1 where the plus and minus signs are used to establish the sign of the numeric value to their right. Thus, in those two expressions, the unary operators have a single operand which is the number to their right.

Finally, keep in mind that in the interactive environment, if an expression is the only thing on a line, the value to which that expression evaluates is shown to the programmer upon hitting return. As was discussed in connection with Listing 1.9, this is not true if the same expression appears in a file. In that case, in order to see the value to which the expression evaluates, the programmer has to use a `print()` statement to explicitly generate output.

2.3 Statements and the Assignment Operator

A *statement* is a complete command (which may or may not produce data). We have already been using `print()` statements. `print()` statements produce output but they do not produce data. At this point this may be a confusing distinction, but we will return to it shortly.

In the previous section we discussed the arithmetic operators that you probably first encountered very early in your education (i.e., addition, subtraction, etc.). What you learned previously about these operators still pertains to how these operators behave in Python. Shortly after being introduced to addition and subtraction, you were probably introduced to the equal sign (=). Unfortunately, ***the equal sign in Python does not mean the same thing as in math!***

In Python (and in many other computer languages), the equal sign is the *assignment operator*. This is a binary operator. When we write the equal sign we are telling Python: "Evaluate the operand/expression on the right side of the equal sign and assign that to the operand on the left." The left operand must be something to which we can assign a value. Sometimes this is called an "lvalue" (pronounced ell-value). In Python we often call this left operand an *identifier* or a *name*. In many other computer languages (and in Python) a more familiar name for the left operand is simply *variable*.[9] You should be familiar with the notion of variables from your math classes. A variable provides a name for a value. The term variable can also imply that something is changing, but this isn't necessarily the case in programs—a programmer may define a variable that is associated with a value that doesn't change throughout a program.

In a math class you might see the following statements: $x = 7$ and $y = x$. In math, how is x related to y if x changes to, say, 9? Since we are told that y is equal to x, we would say that y must also be equal to 9. In general, ***this is not how things behave in computer languages!***

In Python we can write statements that *appear* to be identical to what we would write in a math class, but they are not the same. Listing 2.5 illustrates this. (For brevity, we will use single-character variable names here. The rules governing the choice of variable names are discussed in Sec. 2.6.)

Listing 2.5 Demonstration of the assignment operator.

```
>>> x = 7                         # Assign 7 to variable x.
>>> y = x                         # Assign current value of x to y.
>>> print("x =", x, ", y =", y)   # See what x and y are.
```

[9] In Sec. 2.7 we present more details about what we mean by an *identifier* or a *name* in Python. We further consider the distinction between an lvalue and a variable in Sec. 6.6.

```
4 x = 7 , y = 7
5 >>> x = 9                          # Assign new value to x.
6 >>> print("x =", x, ", y =", y)   # See what x and y are now.
7 x = 9 , y = 7
```

In line 1 we assign the value 7 to the variable x. In line 2, we do *not* equate x and y. Instead, in line 2, we tell Python to evaluate the expression to the right of the assignment operator. In this case, Python merely gets the current value of x, which is 7, and then it assigns this value to the variable on the left, which is the variable y. Line 3 shows that, in addition to literals, the print() function also accepts variables as arguments. The output on line 4 shows us that both x and y have a value of 7. However, although these values are equal, they are distinct. The details of what Python does with computer memory isn't important for us at the moment, but it helps to imagine that Python has a small amount of computer memory where it stores the value associated with the variable x and in a different portion of memory it stores the value associated with the variable y.

In line 5 the value of x is changed to 9, i.e., we have used the assignment operator to assign a new value to x. Then, in line 6, we again print the values of x and y. Here we see that y does not change! So, again, the statement in line 2 does not establish that x and y are equal. Rather, it assigns the current value of x to the variable y. A subsequent change to x or y does not affect the other variable.

Let us consider one other common idiom in computer languages which illustrates that the assignment operator is different from mathematical equality. In many applications we want to change the value of a variable from its current value but the change incorporates information about the current value. Perhaps the most common instance of this is incrementing or decrementing a variable by 1 (we might do this with a variable we are using as a counter). The code in Listing 2.6 demonstrates how one would typically increment a variable by 1. Note that in the interactive environment, if we enter a variable on a line and simply hit return, the value of the variable is echoed back to us. (Python evaluates the expression on a line and shows us the resulting value. When they appear to the right of the assignment operator, variables simply evaluate to their associated value.)

Listing 2.6 Demonstration of incrementing a variable.

```
1 >>> x = 10        # Assign a value to x.
2 >>> x             # Check the value of x.
3 10
4 >>> x = x + 1     # Increment x.
5 >>> x             # Check that x was incremented.
6 11
```

Line 4 of Listing 2.6 makes no sense mathematically: there is no value of x that satisfies the expression if we interpret the equal sign as establishing equality of the left and right sides. However, line 4 makes perfect sense in a computer language. Here we are telling Python to evaluate the expression on the right. Since x is initially 10, the right hand side evaluates to 11. This value is then assigned back to x—this becomes the new value associated with x. Lines 5 and 6 show us that x was successfully incremented.

In some computer languages, when a variable is created, its type is fixed for the duration of the variable's life. In Python, however, a variable's type is determined by the type of the value that has been most recently assigned to the variable. The code in Listing 2.7 illustrates this behavior.

Listing 2.7 A variable's type is determined by the value that is last assigned to the variable.

```
1  >>> x = 7 * 3 * 2
2  >>> y = "is the answer to the ultimate question of life"
3  >>> print(x, y)          # Check what x and y are.
4  42 is the answer to the ultimate question of life
5  >>> x, y                 # Quicker way to check x and y.
6  (42, 'is the answer to the ultimate question of life')
7  >>> type(x), type(y)     # Check types of x and y.
8  (<class 'int'>, <class 'str'>)
9  >>> # Set x and y to new values.
10 >>> x = x + 3.14159
11 >>> y = 1232121321312312312312 * 9873423789237438297
12 >>> print(x, y)          # Check what x and y are.
13 45.14159 12165255965071649871208683630735493412664
14 >>> type(x), type(y)     # Check types of x and y.
15 (<class 'float'>, <class 'int'>)
```

The first two lines of Listing 2.7 set variables `x` and `y` to an integer and a string, respectively. We can see this implicitly from the output of the `print()` statement in line 3 (but keep in mind that just because the output from a `print()` statement appears to be a numeric quantity doesn't mean the value associated with that output necessarily has a numeric type).

The expression in line 5 is something new. This shows that if we separate multiple variables (or expressions) with commas on a line and hit return, the interactive environment will show us the values of these variables (or expressions). This output is slightly different from the output produced by a `print()` statement in that the values are enclosed in parentheses and strings are shown enclosed in quotes.[10] When working with the interactive environment, you may want to keep this in mind for when you want to quickly check the values of variables (and, as we shall see, other things).

In line 7 of Listing 2.7 we use the `type()` function to explicitly show the types of `x` and `y`. In lines 10 and 11 we set `x` and `y` to a `float` and an `int`, respectively. Note that in Python an `int` can have an arbitrary number of digits, limited only by the memory of your computer.

Finally, before concluding this section, we want to mention that we will describe the relationship between a variable and its associated value in various ways. We may say that a variable points to a value, or a variable has a value, or a variable is equal to a value, or simply a variable is a value. As examples, we might say "`x` points to `7`" or "`x` has a value of `7`" or "`x` is `7`". We consider all these statements to be equivalent and discuss this further in Sec. 2.7.

[10]The output produced here is actually a collection of data known as a *tuple*. We will discuss tuples in Sec. 6.6.

2.4 Cascaded and Simultaneous Assignment

There are a couple of variations on the use of the assignment operator that are worth noting. First, it is possible to assign the same value to multiple variables using *cascaded* assignments. Listing 2.8 provides an example. In line 1 the expression on the right side is evaluated. The value of this expression, `11`, is assigned to `x`, then to `y`, and then to `z`. We can cascade any number of assignments but, other than the operand on the extreme right, all the other operands must be lvalues/variables. However, keep in mind that, as discussed in Sec. 2.3, even though `x`, `y`, and `z` are set to the same value, we can subsequently change the value of any one of these variables, and it will not affect the values of the others.

Listing 2.8 An example of cascading assignment in which the variables `x`, `y`, and `z` are set to the same value with a single statement.

```
1 >>> z = y = x = 2 + 7 + 2
2 >>> x, y, z
3 (11, 11, 11)
```

A cascading assignment is a construct that exists in many other computer languages. It is a useful shorthand (allowing one to collapse multiple statements into one), but typically doesn't change the way we think about implementing an algorithm. Cascaded assignments can make the code somewhat harder to read and thus should be used sparingly.

Python provides another type of assignment, known as simultaneous assignment, that is not common in other languages. Unlike cascaded assignments, the ability to do simultaneous assignment *can* change the way we implement an algorithm. With simultaneous assignment we have multiple expressions to the right of the assignment operator (equal sign) and multiple lvalues (variables) to the left of the assignment operator. The expressions and lvalues are separated by commas. For simultaneous assignment to succeed, there must be as many lvalues to the left as there are comma-separated expressions to the right.[11]

Listing 2.9 provides an example in which initially the variable `c` points to a string that is considered the "current" password and the variable `o` points to a string that is assumed to be the "old" password. The programmer wants to exchange the current and old passwords. This is accomplished using simultaneous assignment in line 5.

Listing 2.9 An example of simultaneous assignment where the values of two variables are exchanged with the single statement in line 5.

```
1 >>> c = "deepSecret"          # Set current password.
2 >>> o = "you'll never guess" # Set old password.
3 >>> c, o                      # See what passwords are.
4 ('deepSecret', "you'll never guess")
5 >>> c, o = o, c               # Exchange the passwords.
```

[11]It is also possible to have a single lvalue to the left of the assignment operator and multiple expressions to the right. This, however, is not strictly simultaneous assignment. Rather, it is a regular assignment where a single tuple, which contains the multiple values, is assigned to the variable. We will return to this in Chap. 6.

```
6 >>> c, o                    # See what passwords are now.
7 ("you'll never guess", 'deepSecret')
```

The output generated by the statement in line 6 shows that the values of the current and old passwords have been swapped. There is something else in this code that is worth noting. The string in line 2 includes a single quote. This is acceptable provided the entire string is surrounded by something other than a single quote (e.g., double quotes or another quoting option considered later).

To accomplish a similar swap in a language without simultaneous assignment, we have to introduce a "temporary variable" to hold one of the values while we overwrite one of the variables. Listing 2.10 demonstrates how this swap is accomplished without the use of simultaneous assignment.

Listing 2.10 Demonstration of the swapping of two variables without simultaneous assignment. A temporary variable (here called `t`) must be used.

```
1 >>> c = "deepSecret"          # Set current password.
2 >>> o = "you'll never guess" # Set old password.
3 >>> c, o                      # See what passwords are.
4 ('deepSecret', "you'll never guess")
5 >>> t = c                     # Assign current to temporary variable.
6 >>> c = o                     # Assign old to current.
7 >>> o = t                     # Assign temporary to old.
8 >>> c, o                      # See what passwords are now.
9 ("you'll never guess", 'deepSecret')
```

Clearly the ability to do simultaneous assignment simplifies the swapping of variables and, as we will see, there are many other situations where simultaneous assignment proves useful. But, one should be careful not to overuse simultaneous assignment. For example, one could use simultaneous assignment to set the values of `x` and `y` as shown in line 1 of Listing 2.11. Although this is valid code, it is difficult to read and it would be far better to write two separate assignment statements as is done in lines 6 and 7.

Listing 2.11 Code that uses simultaneous assignment to set the value of two variables. Although this is valid code, there is no algorithmic reason to implement things this way. The assignment in line 1 should be broken into two separate assignment statements as is done in lines 6 and 7.

```
1 >>> # A bad use of simultaneous assignment.
2 >>> x, y = (45 + 34) / (21 - 4), 56 * 57 * 58 * 59
3 >>> x, y
4 (4.647058823529412, 10923024)
5 >>> # A better way to set the values of x and y.
6 >>> x = (45 + 34) / (21 - 4)
7 >>> y = 56 * 57 * 58 * 59
8 >>> x, y
9 (4.647058823529412, 10923024)
```

2.5 Multi-Line Statements and Multi-Line Strings

The end of a line usually indicates the end of a statement. However, statements can span multiple lines if we *escape* the newline character at the end of the line or if we use enclosing punctuation marks, such as parentheses, to span multiple lines. To escape a character is to change its usual meaning. This is accomplished by putting a forward slash (\) in front of the character. The usual meaning of a newline character is that a statement has ended.[12] If we put the forward slash at the end of the line, just before typing return on the keyboard, then we are saying the statement continues on the next line. This is demonstrated in Listing 2.12 together with the use of parentheses to construct multi-line statements.

Listing 2.12 Constructing multi-line statements by either escaping the newline character at the end of the line (as is done in line 1) or by using parentheses (as is done in lines 3 and 5).

```
1 >>> x = 3 + \
2 ... 4
3 >>> y = (123213123123121212312 * 3242342423
4 ... + 2343242332 + 67867687678 - 2
5 ... + 6)
6 >>> x, y
7 (7, 399499136172418158920598441990)
```

Here we see, in lines 2, 4, and 5, that the prompt changes to triple dots when the interpreter is expecting more input.

The converse of having a statement span multiple lines is having multiple statements on a single line. The ability to do this was mentioned in Sec. 1.3 and demonstrated in Listing 1.5. One merely has to separate the statements with semicolons. However, for the sake of maximizing code readability, it is best to avoid putting multiple statements on a line.

Often we want to construct multi-line strings. This can be done in various ways but the simplest technique is to enclose the string in either a single or double quotation mark that is repeated three times.[13] This is illustrated in Listing 2.13.

Listing 2.13 Using triple quotation marks to construct multi-line strings.

```
>>> s = """This string
... spans
... multiple lines."""
>>> t = '''This
... sentence
... no
```

[12]Keep in mind that although we cannot see a newline character, there is a collection of bits at the end of a line that tells the computer to start subsequent output on a new line.

[13]When using triple quotation marks, the string doesn't have to span multiple lines, but since one can use a single set of quotation marks for a single-line string, there is typically no reason to use triple quotation marks for a single-line string.

```
 7 ... verb.'''
 8 >>> print(s)
 9 This string
10 spans
11 multiple lines.
12 >>> print(t)
13 This
14 sentence
15 no
16 verb.
17 >>> s, t   # Quick check of what s and t are.
18 ('This string\nspans\nmultiple lines.', 'This\nsentence\nno\nverb.')
```

In lines 1 through 3 and lines 4 through 7 we create multi-line strings and assign them to variables `s` and `t`. The `print()` statements in lines 8 and 12 show us that `s` and `t` are indeed multi-line strings. In line 17 we ask Python to show us the values of `s` and `t`. The output on line 18 appears rather odd in that the strings do not span multiple lines. Instead, where a new line should start we see `\n`. `\n` is Python's way of indicating a newline character—the `n` is escaped so that it doesn't have its usual meaning. We will consider strings in much more detail in Chap. 9.[14]

2.6 Identifiers and Keywords

In addition to assigning names to values (or objects), there are other entities, such as functions and classes, for which programmers must provide a name. These names are known as *identifiers*. The rules that govern what is and isn't a valid identifier are the same whether we are providing a name for a function or variable or anything else. A valid identifier starts with a letter or an underscore and then is followed by any number of letters, underscores, and digits. No other characters are allowed. Identifiers are case sensitive so that, for example, `ii`, `iI`, `Ii`, and `II`, are all unique names.

Examples of some valid and invalid identifiers are shown in Listing 2.14.

Listing 2.14 Some valid and invalid identifiers.

`one2three`	Valid.
`one_2_three`	Valid.
`1_2_three`	Invalid. Cannot start with a digit.
`one-2-three`	Invalid. Cannot have hyphens (minus signs).
`n31`	Valid.
`n.31`	Invalid. Cannot have periods.
`trailing__`	Valid.
`__leading`	Valid.
`net worth`	Invalid. Cannot have blank spaces.
`vArIaBlE`	Valid.

[14] If you were paying close attention to the output presented in Listing 1.12, you would have noticed `\n` in it.

When choosing an identifier, it is best to try to provide a descriptive name that isn't too long; for example, `area` may be sufficiently descriptive while `area_of_the_circle` may prove cumbersome. There are some identifiers that are valid but, for the sake of clarity, are best avoided. Examples include `l` (lowercase "ell" which may look like one), `O` (uppercase "oh" which may look like zero), and `I` (uppercase "eye" which may also look like one).

It is also good practice to use short variable names consistently. For example, `s` is often used as a name for a string while `ch` is used for a string that consists of a single character (i.e., `ch` connotes character). The variables `i` and `j` are often used to represent integers while `x` and `y` are more likely to be used for `float`s.

In addition to the rules that govern identifiers, there are also *keywords* (sometimes called reserved words) that have special meaning and *cannot* be used as an identifier. All the keywords in Python 3 are listed in Listing 2.15.[15] Keywords will appear in a bold blue `Courier` font.

Listing 2.15 The 33 keywords in Python 3.

```
False     None     True     and      as       assert   break
class     continue def      del      elif     else     except
finally   for      from     global   if       import   in
is        lambda   nonlocal not      or       pass     raise
return    try      while    with     yield
```

At this point we don't know what these keywords represent. The important point, however, is that apart from these words, any identifier that obeys the rules mentioned above is valid.

Perhaps we should take a moment to further consider this list of keywords and think about them in the context of what we have learned so far. Thus far we have used two functions that Python provides: `print()` and `type()`. Notice that neither `print` nor `type` appears in the list of keywords. So, can we have a variable named `print`? The interactive interpreter provides an excellent means to answer this question. Listing 2.16 shows what happens when a value is assigned to the identifier `print`. The statement in line 1 assigns the integer `10` to `print` and we see, in lines 2 and 3, that this assignment works! There is no error. Now we have to wonder: What happened to the `print()` function? In line 4 an attempt is made to print `Hello!` The output following line 4 shows this is not successful—we have lost our ability to print since Python now thinks `print` is the integer `10`!

Listing 2.16 Demonstration of what happens when we assign a value to a built-in Python function.

```
1 >>> print = 10        # Assign 10 to print.
2 >>> print             # Check what print is.  Assignment worked!
3 10
4 >>> print("Hello!")   # Try to print something...
5 Traceback (most recent call last):
```

[15]You can produce this list from within Python by issuing these two commands: `import keyword; print(keyword.kwlist)`.

```
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

Variables spring into existence when we assign them a value. If an identifier had previously been used, the old association is forgotten when a new value is assigned to it. In practice this is rarely a problem. There are very few built-in Python functions that you would consider using as an identifier. There is one possible exception to this that we will discuss in more detail in Chap. 6.[16]

For now, the use of leading underscores is discouraged as Python uses leading underscores for some internal identifiers. Also identifiers with leading underscores are treated slightly differently than other identifiers. However, underscores within an identifier, such as `net_worth`, are quite common and are encouraged.

Finally, there is one final identifier that is worth noting. In the interactive environment (and only in the interactive environment), a single underscore (_) is equal to the value of the most recently evaluated expression. Although we will not use this functionality in this book, it does prove useful when using the interactive environment as a calculator. Listing 2.17 briefly illustrates this.

Listing 2.17 Using "_" to recall the value of previously evaluated expressions.

```
>>> 5 + 4         # Enter an expression.
9
>>> _             # See that _ is equal to value of last expression.
9
>>> _ + 27        # Use _ in new expression.
36
>>> print(_)      # See that _ has been reset.
36
>>> _ - 15
21
>>> result = _    # Store value of _ to a variable.
>>> result
21
```

2.7 Names and Namespaces

Let us now delve deeper into what happens when a value is assigned to a variable. A variable has a name, i.e., an identifier. The assignment serves to associate the value with the name. This value really exists in the form of an *object*. (We aren't concerned here with the details of objects,[17] so we

[16]If you want to learn more now, there is a built-in function called `list()`. Lists are commonly used in Python and it is tempting to use `list` as an identifier of your own. You can do this, but you will then lose easy access to the built-in function `list()`. However, one further aside: If you ever mask one of the built-in functions by assigning a value to the function's identifier, you can recover the built-in function by deleting, using the `del()` function, your version of the identifier. Returning to Listing 2.16 as an example, to recover the built-in `print()` function you would issue the statement `del(print)`. Note that `del` is a keyword and hence you cannot accidently assign it a new value.

[17]Objects are discussed in Chap. 5.

can simply think of an object as a "thing" that has a value and we are only interested in this value. For example, although the integer `7` is really treated by Python as a particular kind of object, here we consider this to be simply the integer `7`.)

Within your computer's memory, Python maintains what are known as *namespaces*. A namespace is used to keep track of all the currently defined names (identifiers) as well as the objects associated with these names. If, within an expression, you refer to a name that was *not* previously defined, an exception will be raised (i.e., an error will occur). However, if a previously undefined name is used on the *left* side of the assignment operator, then the name is created within the namespace and it is associated with the value produced by the expression to the right of the assignment operator.

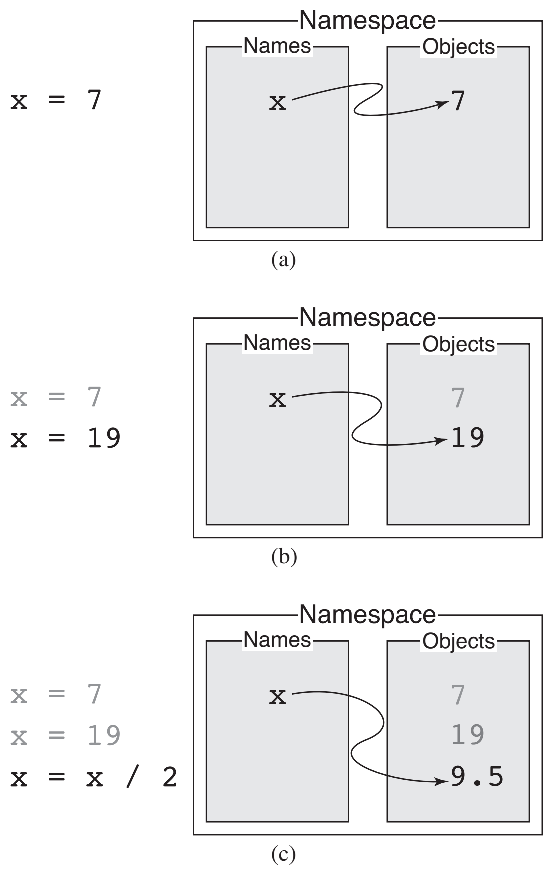
The behavior of namespaces is illustrated in Fig. 2.1. The namespace is shown to the right of the figure and the code that is assumed to be executed is shown to the left. In the figure, the namespace is depicted as consisting of two parts: a list of names and a list of objects (where only the values of the objects are shown). The namespace serves to map names to objects. In Fig. 2.1(a), the statement `x = 7` is executed. The name to the left side of the assignment operator, `x`, is placed in the list of names and is associated with the value `7`. This association is indicated by the curved line. We can think of this as "`x` points to `7`." In Fig. 2.1(b), the statement `x = 19` is executed. (Here we are assuming this statement is executed in the same session in which the statement `x = 7` was previously executed.) Since the name `x` already exists in the list of names, no new name is created. Instead, an object with a value of `19` is created and `x` is associated with this new value. You may ask: What happens to the value `7`? The fate of that object is up to Python and not our concern. In the figure, rather than removing the object from the list of objects, we show that it persists but no name is associated with it (i.e., nothing points to it and hence it is shown in gray rather than black).

Finally, in Fig. 2.1(c), the statement `x = x / 2` is executed. Recall that in assignment statements the expression to the right of the assignment operator is evaluated and then this value is associated with the name to the left of the assignment operator. Here `x` appears in the expression to the right of the assignment operator. This does not cause a problem since `x` has previously been defined. When this expression is evaluated, the value associate with `x` is `19`. The result of `19 / 2` is the `float` value `9.5`, hence after this third statement is executed, `x` is associated with the `float 9.5`.

As another example of the behavior of namespaces, consider Fig. 2.2 which involves two variables, `x` and `y`. Listing 2.2(a) is no different from Listing 2.1(a): the name `x` is assigned the value `7`. Figure 2.1(b) depicts what happens when the statement `y = x` is executed. From your math classes, it would be natural to think this statement would somehow associate `y` directly with `x`. But, this is *not* the case. Instead, as always, the right hand side is evaluated and the resulting value is associated with the name `y`. In this case, the right hand side simply evaluates to `7` and `y` is associated with that. Since `7` is already in the list of objects, Python does not create a new object but rather has `y` point to the same object as `x`.

Figure 2.1(c) depicts what happens when the statement `x = 19` is executed. Python does not change the value of an integer object, i.e., it will not change the value in memory from `7` to `19`. Instead, it creates a new object and associates `x` with it.

As a final example, Fig. 2.3 depicts the behavior of a namespace when code involving simultaneous assignment is executed. In Fig. 2.3(a), `x` and `y` are simultaneously assigned values: `x` is assigned the integer `2` while `y` is assigned the string `two`. In Fig. 2.3(b), `x` is assigned a new


```
x = 7
```

```
x = 7
x = 19
```

```
x = 7
x = 19
x = x / 2
```

Figure 2.1: Depiction of a namespace which consists of names (i.e., identifiers or variables) and a collection of objects (where only the value of the object is shown). (a) Contents of the namespace after the statement `x = 7` is executed. (b) Contents of the namespace after executing `x = 19`. (c) Contents of the namespace after executing `x = x / 2`.

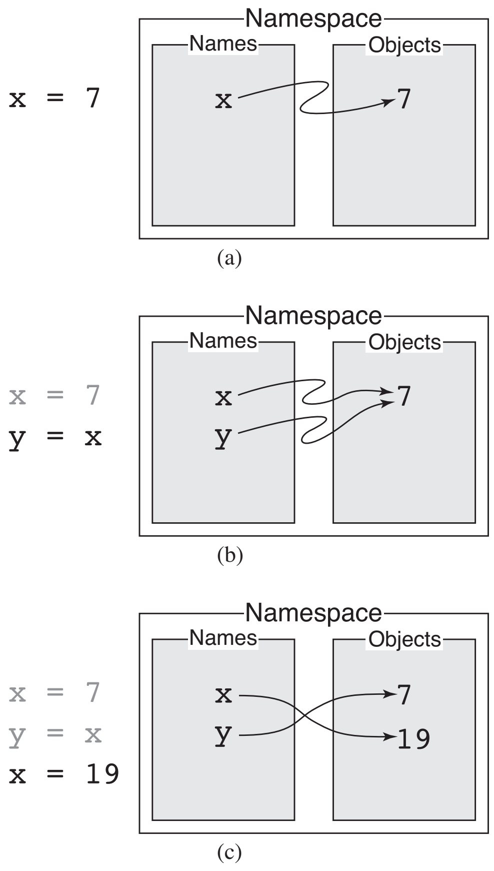


Figure 2.2: Depiction of another namespace (i.e., independent of the namespace shown in Fig. 2.1). (a) Contents of the namespace after the statement `x = 7` is executed. (b) Contents of the namespace after executing `y = x`. (c) Contents of the namespace after executing `x = 19`.

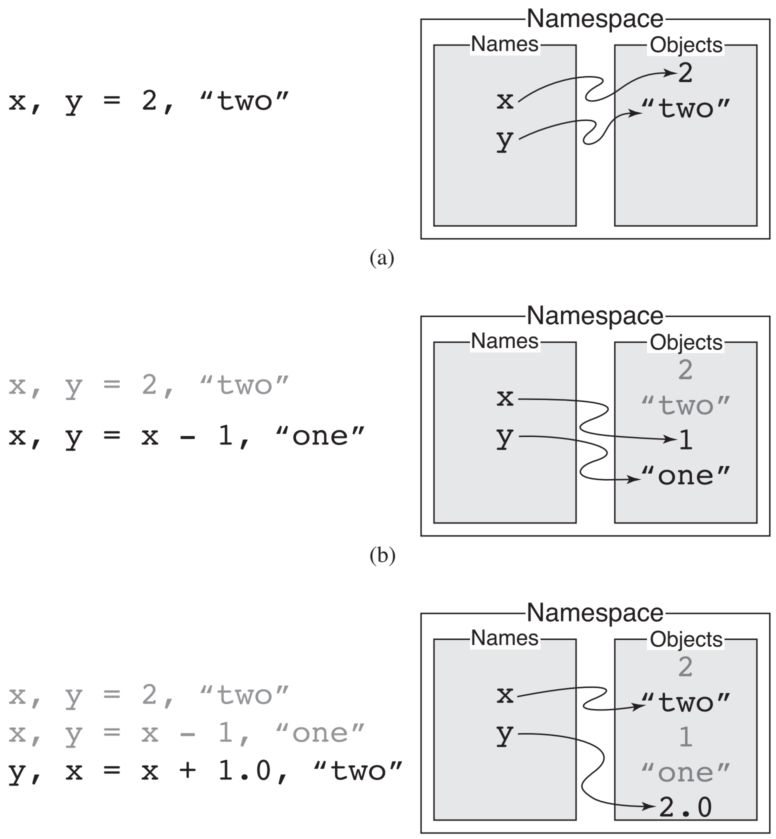


Figure 2.3: Depiction of another namespace. The code that is executed involves simultaneous assignment to the names `x` and `y`. (a) `x` and `y` are initialized to an integer and a string, respectively. (b) `x` and `y` are assigned different integer and string values. (c) `y` is set to a `float` (based on the current value of `x` and `x` is assigned a `string`.

integer value (that is obtained by decrementing its initial value by one) and `y` is assigned the string `one`. In Fig. 2.3(c), `y` is assigned the `float 2.0` (which is obtained by adding `1.0` to the current value of `x`) and `x` is assigned the string `two`. Since the `float 2.0` is distinct from the integer `2`, in Fig. 2.3(c) the value `2.0` is added to the list of objects.

Again we note that, as Figs. 2.1–2.3 illustrate, in Python a name can, at different points in a program, be associated with values of different type. Thus, a given name may be associated with an integer in one section of the code, with a `float` in a different section of the code, and a string in yet another section. This “freedom” to easily associate names with different types of values is *not* possible in many other computer languages. For example, in C a variable can typically only be associated with values of a fixed type (the value itself can change, but its type cannot). This ability in Python is generally a blessing but can be something of a curse. It allows the programmer to express things in a few lines of code that might otherwise take many more. However, it can be a source of bugs when the underlying type isn't what the programmer assumes.

2.8 Additional Arithmetic Operators

There are other arithmetic operators in Python that are somewhat less common than those discussed in Sec. 2.2, but are nevertheless quite important in the implementation of many algorithms.

2.8.1 Exponentiation

Exponentiation is obtained with the operator `**`. The expression `x ** y` says to calculate x^y. This operator is unlike the other arithmetic operators in that it associates (or evaluates) right to left. The expression `x ** y ** z` is equivalent to $x^{y^z} = x^{(y^z)}$ and *not* $(x^y)^z$. If both operands are `int`s, the result is an `int`. If either operand is a `float`, the result is a `float`. Listing 2.18 demonstrates the use of exponentiation.

Listing 2.18 Demonstration of the `**` (exponentiation) operator.

```
1  >>> 2 ** 3
2  8
3  >>> 2 ** 3 ** 4     # Operator is right associative.
4  2417851639229258349412352
5  >>> 3 ** 4
6  81
7  >>> (2 ** 3) ** 4   # Use parentheses to change order of operation.
8  4096
9  >>> 3 ** 400        # A big number.  Exact value as an int.
10 70550791086553325712464271575934796216507949612787315762871223209262085
11 55158293415657929852944713415815495233482535591186692979307182456669414
12 5084454535257027960285323760313192443283334088001
13 >>> 3.0 ** 400      # A big number.  Approximate value as a float.
14 7.055079108655333e+190
```

In line 1 we calculate 2^3 and in line 3 we calculate $2^{3^4} = 2^{81}$. In line 9 3^{400} is calculated using integers. The result shown on the next three lines is the exact result; this number is actually returned as one continuous number (i.e., without explicit line breaks) but here the number has been split into three lines for the sake of readability. Lines 13 and 14 show what happens when one of the arguments is a `float`. The result in line 14 is an approximation to the exact answer. The `float` result has about 16 digits of precision whereas the exact result has 191 digits.

2.8.2 Floor Division

Recall from Sec. 2.2 that `float` division is represented by the operator `/`. (If one simply says "division," `float` division is implied.) Python provides another type of division which is known as floor division. Many computer languages provide two types of division and Python's floor division is similar to what is known as integer division in most other languages. In most other languages, both integer and `float` division share the same symbol, i.e., `/`. The type of division that is done depends on the types of the two operands involved. This can be confusing and often leads to bugs! Python is different in that the operator `/` *always* represents `float` division while `//` is *always* used for floor division. In floor division the quotient is a whole number. Specifically, the result is the greatest whole number that does not exceed the value that would be obtained with float division. For example, if the value obtained from `float` division was `3.2`, the result from floor division would be either the integer `3` or the `float` `3.0` since three is the largest whole number that doesn't exceed `3.2`. The result is an `int` if both operands are integers but is a `float` if either or both operands are `floats`. As another example, if the value obtained from `float` division was `-3.2`, the result from floor division would be either the integer `-4` or the `float` `-4.0` since negative four is the largest whole number that doesn't exceed `-3.2`.

Floor division may not seem to be a useful tool in that it discards information (the fractional part). However, in many problems it only makes sense to talk about whole units of a quantity. Thus, discarding the fractional part is appropriate. For example, what if we want to know how many complete minutes there are in 257 seconds or how many quarters there are in 143 pennies? The first six lines of Listing 2.19 illustrate how floor division answers these questions.

Listing 2.19 Demonstration of floor division.

```
1  >>> time = 257              # Time in seconds.
2  >>> minutes = time // 60 # Number of complete minutes in time.
3  >>> print("There are", minutes, "complete minutes in", time, "seconds.")
4  There are 4 complete minutes in 257 seconds.
5  >>> 143 // 25
6  5
7  >>> 143.4 // 25
8  5.0
9  >>> 9 // 2.5
10 3.0
```

Lines 7 through 10 show that `floats` can be used with floor division. The result is a `float` but with a fractional part of zero (i.e., a whole number represented as a `float`).

Floor division has the same precedence as multiplication and `float` division.

2.8.3 Modulo and `divmod()`

Another arithmetic operator that you may not have encountered before is the *modulo*, or sometimes called *remainder*, operator. In some sense, modulo is the complement of floor division. It tells us what remains after the left operand is divided by the right operand an integer number of times. The symbol for modulo is `%`. An expression such as `17 % 5` is read as seventeen mod five (this expression evaluates to `2` since `5` goes into `17` three times with a remainder of `2`). Although modulo may be new to you, it is extremely useful in a wide range of algorithms. To give a hint of this, assume we want to express 257 seconds in terms of minutes and seconds. From Listing 2.19 we already know how to obtain the number of minutes using floor division. How do we obtain the number of seconds? If we have the number of minutes, we can calculate the remaining number of seconds as line 3 of Listing 2.20 shows.

Listing 2.20 Slightly cumbersome way to find the number of minutes and seconds in a given time (which is assumed to be in seconds).

```
1 >>> time = 257                    # Initialize total number of seconds.
2 >>> minutes = time // 60          # Calculate minutes.
3 >>> seconds = time - minutes * 60 # Calculate seconds.
4 >>> minutes, seconds              # Show minutes and seconds.
5 (4, 17)
```

In Listing 2.20 we first calculate the minutes and then, using that result, calculate the number of seconds. However, what if we just wanted the number of seconds? We can obtain that directly using the modulo operator, i.e., we can replace the statement in line 3 with `seconds = time % 60`.

Modulo has the same operator precedence as division and multiplication.

If we want to calculate both floor division and modulo with the same two operands, we can use the function `divmod()`. This function returns the results of both floor division and modulo. Thus,

```
divmod(x, y)
```

is equivalent to

```
x // y, x % y
```

We can use simultaneous assignment to assign the two results to two variables. Listing 2.21 demonstrates this.

Listing 2.21 Using the `divmod()` function to calculate both floor division and modulo simultaneously.

```
>>> time = 257        # Initialize time.
>>> SEC_PER_MIN = 60 # Use a "named constant" for 60.
>>> divmod(time, SEC_PER_MIN)   # See what divmod() returns.
(4, 17)
>>> # Use simulataneous assignment to obtain minutes and seconds.
>>> minutes, seconds = divmod(time, SEC_PER_MIN)
>>> # Attempt to display the minutes and seconds in "standard" form.
>>> print(minutes, ":", seconds)
4 : 17
>>> # Successful attempt to display time "standard" form.
>>> print(minutes, ":", seconds, sep="")
4:17
>>> # Obtain number of quarters and leftover change in 143 pennies.
>>> quarters, cents = divmod(143, 25)
>>> quarters, cents
(5, 18)
```

In addition to illustrating the use of the `divmod()` function, there are two other items to note in Listing 2.21. First, in Listings 2.19 and 2.20 the number `60` appears in multiple expressions. We know that in this context `60` represents the number of seconds in a minute. But, the number `60` by itself has very little meaning (other than the whole number that comes after 59). `60` could represent any number of things: the age of your grandmother, the number of degrees in a particular angle, the weight in ounces of your favorite squirrel, etc. When a number appears in a program without its meaning being fully specified, it is known as a "magic number." There are a few different definitions for magic numbers. The one relevant to this discussion is: "Unique values with unexplained meaning or multiple occurrences which could (preferably) be replaced with named constants."[18] In line 2 of Listing 2.21 we create a *named constant*, `SEC_PER_MIN`, as a substitute for the number `60` in the subsequent code. This use of named constants can greatly enhance the readability of a program. It is a common practice for a named constant identifier to use uppercase letters.

The other item to note in Listing 2.21 pertains to the `print()` statements in lines 8 and 11. In line 8 the `print()` statement has three arguments: the number of minutes, a string (corresponding to a colon), and the number of seconds. When displaying a time, we often separate the number of minutes and seconds with a colon, e.g., `4:17`. However, the output on line 9 isn't quite right. There are extraneous spaces surrounding the colon. The `print()` statement on line 11 fixes this by setting the optional parameter `sep` to an empty string. If you refer to the output shown in Listing 1.12, you will see that, by default, `sep`, which is the separator that appears between the values that are printed, has a value of one blank space. By setting `sep` to an empty string we get the desired output shown in line 12.

As with floor division, if both operands are integers, modulo returns an integer. If either or both operands are a `float`, the result is a `float`. This behavior holds true of the arguments of the `divmod()` function as well: both arguments must be integers for the return values to be integers.

2.8.4 Augmented Assignment

As mentioned in Sec. 2.3, it is quite common to assign a new value to a variable that is based on the old value of that variable, e.g., `x = x + 1`. In fact, this type of operation is so common that many languages, including Python, provide arithmetic operators that serve as a shorthand for this. These are known as *augmented assignment operators.* These operators use a compound symbol consisting of one of the "usual" arithmetic operators together with the assignment operator. For example, `+=` is the addition augmented assignment operator. Other examples include `-=` and `*=` (which are the subtraction augmented assignment operator and the multiplication augmented assignment operator, respectively).

To help explain how these operators behave, let's write a general augmented assignment operator as *<op>=* where *<op>* is a placeholder for one of the usual arithmetic operators such `+`, `-`, or `*`. A general statement using an augmented assignment operator can be written as

```
<lvalue> <op>= <expression>
```

[18] `http://en.wikipedia.org/wiki/Magic_number_(programming)`

where `<lvalue>` is a variable (i.e., an lvalue), `<op>`= is an augmented assignment operator, and `<expression>` is an expression. The following is a specific example that fits this general form

```
x += 1
```

Here `x` is the lvalue, += is the augmented assignment operator, and the literal `1` is the expression (albeit a very simple one—literals are the simplest form of an expression). This statement is completely equivalent to

```
x = x + 1
```

Now, returning to the general expression, an equivalent statement in non-augmented form is

```
<lvalue> = <current_value_of_lvalue> <op> (<expression>)
```

where `<current_value_of_lvalue>` is, naturally, the value of the lvalue prior to the assignment. The parentheses have been added to `<expression>` to emphasize the fact that this expression is completely evaluated before the operator associated with the augmented assignment comes into play.

Let's consider a couple of other examples. The following statement

```
y -= 2 * 7
```

is equivalent to

```
y = y - (2 * 7)
```

while

```
z *= 2 + 5
```

is equivalent to

```
z = z * (2 + 5)
```

Note that if we did not include the parentheses in this last statement, it would not be equivalent to the previous one.

There are augmented assignment versions of all the arithmetic operators we have considered so far. Augmented assignment is further illustrated in the code in Listing 2.22.

Listing 2.22 Demonstration of the use of augmented assignment operators.

```
>>> x = 22        # Initialize x to 22.
>>> x += 7        # Equivalent to: x = x + 7
>>> x
29
>>> x -= 2 * 7    # Equivalent to: x = x - (2 * 7)
>>> x
15
>>> x //= 5       # Equivalent to: x = x // 5
>>> x
```

```
3
>>> x *= 100 + 20 + 9 // 3 # Equivalent to: x = x * (100 + 20 + 9 // 3)
>>> x
369
>>> x /= 9         # Equivalent to: x = x / 9
>>> x
41.0
```

2.9 Chapter Summary

Literals are data that are entered directly into the code.

Data has *type*, for example, **int** (integer), **float** (real number with finite precision), and **str** (string, a collection of characters).

type() returns the type of its argument.

A literal float has a decimal point or contains the power-of-ten exponent indicator e, e.g., 1.1e2, 11e1, and 110.0 are equivalent floats.

A literal int does not contain a decimal point (nor the power-of-ten exponent indicator e).

A literal str is enclosed in either a matching pair of double or single quotes. Quotation marks can be repeated three times at the beginning and end of a string, in which case the string can span multiple lines.

\n is used in a string to indicate the newline character.

Characters in a string may be *escaped* by preceding the character with a backslash. This causes the character to have a different meaning than usual, e.g., a backslash can be placed before a quotation mark in a string to prevent it from indicating the termination of the string; the quotation mark is then treated as part of the string.

An *expression* produces data. The simplest expression is a literal.

There are numerous arithmetic operators. *Binary operators* (which require two operands) include:

+, -	⇔	addition and subtraction
/, //	⇔	float and floor division
%	⇔	modulo (or remainder)
*	⇔	multiplication
**	⇔	exponentiation

float division (/) yields a float regardless of the types of the operands. For all other arithmetic operators, only when *both* operands are integers does the operation yield an integer. Said another way, when either or both operands are floats, the arithmetic operation yields a float.

Floor division (//) yields a whole number (which may be either an int or a float, depending on the operands). The result is the largest whole number that does not exceed the value that would be obtained with float division.

Modulo (%) yields the remainder (which may be either an int or a float, depending on the operands) after floor division has been performed.

divmod(a, b): equivalent to ⇒ `a // b, a % b`.

Evaluation of expressions containing multiple operations follows the rules of *precedence* to determine the order of operation. Exponentiation has highest precedence; multiplication, integer and `float` division, and modulo have equal precedence which is below that of exponentiation; addition and subtraction have equal precedence which is below that of multiplication, division, and modulo.

Operations of equal precedence are evaluated left to right except for exponentiation operations which are evaluated right to left.

The negative sign (−) and positive sign (+) can be used as *unary operators*, e.g., `-x` changes the sign of `x`. The expression `+x` is valid but has no effect on the value of `x`.

Parentheses can be used to change the order or precedence.

Statements are complete commands.

The equal sign = is the *assignment operator*. The value of the expression on the right side of the equal sign is assigned to the *lvalue* on the left side of the equal sign.

An *lvalue* is a general name for something that can appear to the left side of the assignment operator. It is typically a variable that must be a *valid identifier*.

A variable can also be thought of as a *name* within a *namespace*. A namespace maps names to their corresponding values (or objects).

Valid identifiers start with a letter or underscore followed by any number of letters, digits, and underscores.

There are 33 *keywords* that cannot be used as identifiers.

Augmented operators can be used as shorthand for assignment statements in which an identifier appears in an arithmetic operation on the left side of the equal sign *and* on the the right side of the assignment operator. For example, `x += 1` is equivalent to `x = x + 1`.

Simultaneous assignment occurs when multiple comma-separated expressions appear to the right side of an equal sign and an equal number of comma-separated lvalues appear to the left side of the equal sign.

A statement can span *multiple lines* if it is enclosed in parentheses or if the newline character at the end of each line of the statement (other than the last) is escaped using a backslash.

Multiple statements can appear on one line if they are separated by semicolons.

A *magic number* is a numeric literal whose underlying meaning is difficult to understand from the code itself. *Named constants* should be used in the place of magic numbers.

2.10 Review Questions

1. What does Python print as a result of this statement:

```python
print(7 + 23)
```

   (a) `7 + 23`

   (b) `7 + 23 = 30`

(c) `30`

(d) This produces an error.

2. Which of the following *are* valid variable names?

   (a) `_1_2_3_`

   (b) `ms.NET`

   (c) `WoW`

   (d) `green-day`

   (e) `big!fish`

   (f) `500_days_of_summer`

3. Which of the following *are* valid identifiers?

   (a) `a1b2c`

   (b) `1a2b3`

   (c) `a_b_c`

   (d) `_a_b_`

   (e) `a-b-c`

   (f) `-a-b-`

   (g) `aBcDe`

   (h) `a.b.c`

4. Suppose the variable `x` has the value `5` and `y` has the value `10`. After executing these statements:

   ```
   x = y
   y = x
   ```

   what will the values of `x` and `y` be, respectively?

   (a) `5` and `10`

   (b) `10` and `5`

   (c) `10` and `10`

   (d) `5` and `5`

5. True or False: "`x ** 2`" yields the identical result that is produced by "`x * x`" for all integer and `float` values of `x`.

6. True or False: "`x ** 2.0`" yields the identical result that is produced by "`x * x`" for all integer and `float` values of `x`.

7. What does Python print as a result of this statement?

```
print(5 + 6 % 7)
```

(a) This produces an error.

(b) `5 + 6 % 7`

(c) `11`

(d) `4`

8. What is the output from the `print()` statement in the following code?

```
x = 3 % 4 + 1
y = 4 % 3 + 1
x, y = x, y
print(x, y)
```

(a) `2 4`

(b) `4 2`

(c) `0 3`

(d) `3 0`

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

```
x = 3
y = 4
print("x", "y", x + y)
```

(a) `3 4 7`

(b) `x y 7`

(c) `x y x + y`

(d) `3 4 x + y`

(e) `x y 34`

For each of the following, determine the value to which the expression evaluates. (Your answer should distinguish between `float`s and `int`s by either the inclusion or exclusion of a decimal point.)

10. `5.5 - 11 / 2`

11. `5.5 - 11 // 2`

12. `10 % 7`

13. `7 % 10`

14. `3 + 2 * 2`

15. `16 / 4 / 2`

16. `16 / 4 * 2`

17. Given that the following Python statements are executed:

```
a = 2
b = a + 1 // 2
c = a + 1.0 // 2
d = (a + 1) // 2
e = (a + 1.0) // 2
f = a + 1 / 2
g = (a + 1) / 2
```

    Determine the values to which the variables `b` through `g` are set.

18. What output is produced when the following code is executed?

```
hello = "yo"
world = "dude"
print(hello, world)
```

    (a) `hello, world`
    (b) `yo dude`
    (c) `"yo" "dude"`
    (d) `yodude`
    (e) This produces an error.

19. The following code is executed. What is the output from the `print()` statement?

```
x = 15
y = x
x = 20
print(y)
```

    (a) `15`
    (b) `20`
    (c) `y`
    (d) `x`
    (e) This produces an error.

20. The following code is executed. What is the output from the `print()` statement?

```
result = "10" / 2
print(result)
```

    (a) `5`
    (b) `5.0`
    (c) `'"10" / 2'`
    (d) This produces an error.

21. The following code is executed. What is the output from the `print()` statement?

```
x = 10
y = 20
a, b = x + 1, y + 2
print(a, b)
```

    (a) `10 20`
    (b) `11 22`
    (c) `'a, b'`
    (d) `'x + 1, y + 2'`
    (e) This produces an error.

22. True or False: In general, "`x / y * z`" is equal to "`x / (y * z)`".

23. True or False: In general, "`x / y ** z`" is equal to "`x / (y ** z)`".

24. True or False: In general, "`w + x * y + z`" is equal to "`(w + x) * (y + z)`".

25. True or False: In general, "`w % x + y % z`" is equal to "`(w % x) + (y % z)`".

26. True or False: If both `m` and `n` are `int`s, then "`m / n`" and "`m // n`" both evaluate to `int`s.

27. True or False: The following three statements are all equivalent:

```
x = (3 +
4)

x = 3 + \
4

x = """3 +
4"""
```

28. Given that the following Python statements are executed:

```
x = 3 % 4
y = 4 % 3
```

What are the values of `x` and `y`?

29. To what value is the variable `z` set by the following code?

```
z = 13 + 13 // 10
```

(a) `14.3`

(b) `14.0`

(c) `2`

(d) `14`

(e) `16`

30. Assume the float variable `ss` represents a time in terms of seconds. What is an appropriate statement to calculate the number of complete minutes in this time (and store the result as an `int` in the variable `mm`)?

(a) `mm = ss // 60`

(b) `mm = ss / 60`

(c) `mm = ss % 60`

(d) `mm = ss * 60`

31. To what values does the following statement set the variables `x` and `y`?

```
    x, y = divmod(13, 7)
```

(a) `6` and `1`

(b) `1` and `6`

(c) `6.0` and `2.0`

(d) This produces an error.

ANSWERS: 1) c; 2) a and c are valid; 3) a, c, d, and g are valid; 4) c; 5) True; 6) False, if `x` is an integer `x * x` yields an integer while `x ** 2.0` yields a `float`; 7) c; 8) b; 9) b; 10) `0.0`; 11) `0.5`; 12) `3`; 13) `7`; 14) `7`; 15) `2.0`; 16) `8.0`; 17) `b = 2`, `c = 2.0`, `d = 1`, `e = 1.0`, `f = 2.5`, `g = 1.5`; 18) b; 19) a; 20) d; 21) b; 22) False (the first expression is equivalent to "`x * z / y`"); 23) True; 24) False; 25) True; 26) False ("`m / n`" evaluates to a `float`); 27) False (the first two are equivalent arithmetic expressions, but the third statement assigns a string to `x`); 28) `3` and `1`; 29) d; 30) a; 31) b.

2.11 Exercises

1. Write a small program that assigns an angle in degrees to a variable called `degrees`. The program converts this angle to radians and assigns it to a variable called `radians`. To convert from degrees to radians, use the formula $\text{radians} = \text{degrees} \times 3.14/180$ (where we are using 3.14 to approximate π). Print the angle in both degrees and radians.

   The following demonstrates the program output when the angle is 150 degrees:

```
Degrees:  150
Radians:  2.6166666666666667
```

2. Write a program that calculates the average score on an exam. Assume we have a small class of only three students. Assign each student's score to variables called `student1`, `student2`, and `student3` and then use these variables to find the average score. Assign the average to a variable called `average`. Print the student scores and the average score.

   The following demonstrates the program output when the students have been assigned scores of 80.0, 90.0, and 66.5:

```
Student scores:
80.0
90.0
66.5
Average:  78.83333333333333
```

3. Imagine that you teach three classes. These classes have 32, 45, and 51 students. You want to divide the students in these classes into groups with the same number of students in each group but you recognize that there may be some "left over" students. Assume that you would like there to be 5 groups in the first class (of 32 students), 7 groups in the second class (of 45 students), and 6 groups in the third class (of 51 students). Write a program that uses the `divmod()` function to calculate the number of students in each group (where each group has the same number of students). Print this number for each class and also print the number of students that will be "leftover" (i.e., the number of students short of a full group). Use simultaneous assignment to assign the number in each group and the "leftover" to variables.

   The following demonstrates the program's output:

```
Number of students in each group:
Class 1:  6
Class 2:  6
Class 3:  8
Number of students leftover:
Class 1:  2
Class 2:  3
Class 3:  3
```

4. The Python statements below have several errors. Identify the errors and correct them so that the program properly calculates the circumference of Jimmy's pie ($\text{circumference} = 2\pi r$).

```
pi = '3.14'
pie.diameter = 55.4
pie_radius = pie.diameter // 2
circumference = 2 * pi ** pie_radius
circumference-msg = 'Jimmy's pie has a circumference: '
print(circumference-msg, circumference)
```

   The following demonstrates the output from the corrected program:

```
Jimmy's pie has a circumference:  173.956
```

5. Write a program that calculates the wavelength of a wave traveling at a constant velocity given the speed and the frequency. Use the formula $\lambda = v/f$, where λ (lambda) is wavelength in meters, v is velocity in meters per second, and f is frequency in Hertz (cycles per second). Print the velocity, frequency, and wavelength. Assign each of these values to a variable and use the variables in your `print()` statements.

   The following demonstrates what the program prints:

```
The speed (m/s):  343
The frequency (Hz):  256
The wavelngth (m):  1.33984375
```