3
VARIABLES AND CALCULATIONS

Now you're ready to learn your first elements of Python and start learning how to solve programming problems. Although programming languages have myriad features, the core parts of any programming language are the instructions that perform numerical calculations. In this chapter, we'll explore how math is performed in Python programs and learn how to solve some problems using only mathematical operations.

Sample Program

Let's start by looking at a very simple problem and its Python solution.

PROBLEM: THE TOTAL COST OF TOOTHPASTE

A store sells toothpaste at $1.76 per tube. Sales tax is 8 percent. For a user-specified number of tubes, display the cost of the toothpaste, showing the subtotal, sales tax, and total, including tax.

First I'll show you a program that solves this problem:

toothpaste.py

```
tube_count = int(input("How many tubes to buy: "))

toothpaste_cost = 1.76
subtotal = toothpaste_cost * tube_count
sales_tax_rate = 0.08
sales_tax = subtotal * sales_tax_rate
total = subtotal + sales_tax

print("Toothpaste subtotal: $", subtotal, sep = "")
print("Tax: $", sales_tax, sep = "")
print("Total is $", total, " including tax.", sep = ")
```

Parts of this program may make intuitive sense to you already; you know how you would answer the question using a calculator and a scratch pad, so you know that the program must be doing something similar. Over the next few pages, you'll learn exactly what's going on in these lines of code. For now, enter this program into your Python editor exactly as shown and save it with the required *.py* extension. Run the program several times with different responses to the question to verify that the program works. Over the rest of this chapter, we'll look at the elements of Python that are used in simple programs like this.

Variables, Types, and Mathematics in Python

The variable is one of the most important concepts in high-level programming. To understand the concept, let's start with a line of Python code that uses a variable:

```
first_variable = 37
```

From a glance, it looks like this code associates the word `first_variable` with the number 37 in some way, but what is it actually doing? The simplest way to understand variables is with a spreadsheet program. I'll be using Google Sheets for this explanation, which is a free browser-based application, but any spreadsheet program will work if you want to follow along.

Spreadsheet Cells and Program Variables

Starting with a blank spreadsheet, place the number 37 in cell B2, the second column of the second row. Now we have a particular number (37) in a location that has a particular name (B2), as shown in Figure 3-1.

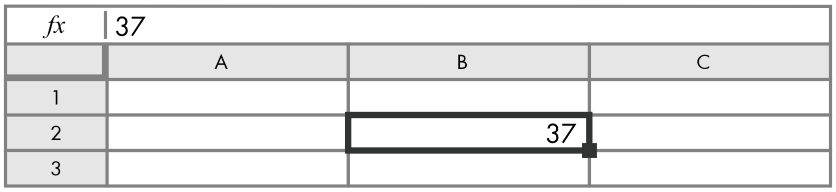


Figure 3-1: Cell B2 has the value 37.

In programs, a *variable* is a location to store data, just like a spreadsheet cell. The data stored in the location, like the 37 in this cell, is the variable's *value*. The name that we use to refer to the location, like the B2 label of our cell, is the variable's *identifier*, or more colloquially, its *name*.

Let's look again at that line of code:

```
first_variable = 37
```

This is an *assignment statement*, a line of code that places the value specified on the right of an equal sign (`=`) in the location specified by an identifier on the left. Here, `first_variable` is our variable identifier, and this assignment statement places the value 37 into it. Put another way, this line of code *assigns* 37 to `first_variable`.

Choosing Variable Identifiers

In Python, variable identifiers are single words made up of letters, digits, and underscores, with the additional rule that the first character of an identifier cannot be a digit. Table 3-1 lists sample valid and invalid Python identifiers.

Table 3-1: Valid and Invalid Python Identifiers

Valid	Invalid
total	1995_Sales
totalSales	total$
total_sales_percent	total_Sales%
quarter3target	
_sales1995	

Any of the styles shown in the left column of the table are valid in Python. In this book, variable identifiers are all lowercase, with underscores between words we would write separately in plain language, as in `first_variable`. This is not because I personally prefer this style. Rather, this is the rule for variable identifiers in *PEP 8*, a quasi-official style guide for Python programs.

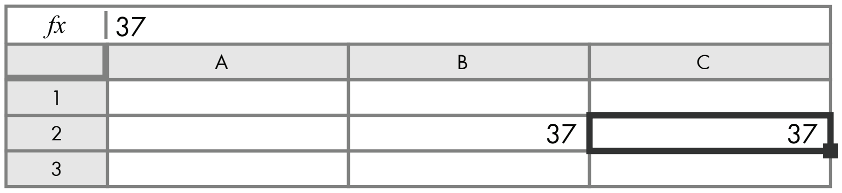
Programming style guides cover issues like how to name variables and how to lay out source code on the page, and help maintain a consistent look throughout a program. Style guides are very useful when multiple people are working on the same program. If you continue programming, you may be required to follow style guides given to you by employers or professors. Also, when expanding an existing program written by someone else, you will usually want to adopt the existing style of the program.

The Python community tends to adhere to PEP 8, and for this reason, the code in this book will follow the guide as much as possible. If you plan on sharing any of your code with other Python programmers on the web, you will spare yourself some trouble if you, too, adopt the PEP 8 guidelines, which can be found at *http://www.python.org/dev/peps/pep-0008/*.

It's important to note, though, that PEP 8, like every other style guide, simply represents the preferences of its author, and there is no "right" answer for naming variables or any other style issue in programming. The downside of using a style guide as a beginner is that you aren't encouraged to think about the choices you are making, so you may not develop the ability to write code in a good style on your own. Whether you follow a style guide or make your own path, just remember the goal of a coding style is a program that's easy to read and understand, and make sure you apply your chosen style consistently.

Copying Values Between Variables

Returning to our spreadsheet, select cell B2, copy the number inside, and paste it into cell C2. The result is shown in Figure 3-2.

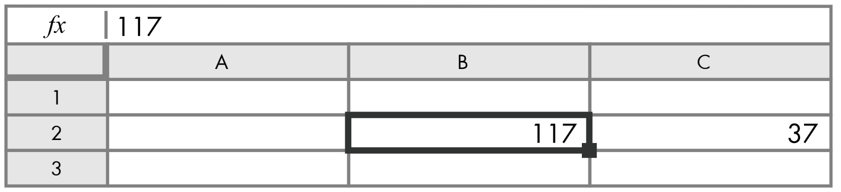


fx 37

	A	B	C
1			
2		37	37
3			

Figure 3-2: Cell C2 is a copy of B2.

Both cells, B2 and C2, now display the same number, 37. If we change the value of B2 from 37 to 117, the value of C2 remains unchanged, as shown in Figure 3-3. The values in the two cells, B2 and C2, are independent.



fx 117

	A	B	C
1			
2		117	37
3			

Figure 3-3: Changing the value in cell B2 doesn't affect C2.

The same idea holds true for variables. Consider the following program:

```
first_variable = 37
second_variable = first_variable
first_variable = 117
```

The first statement we've already seen: it creates `first_variable` and assigns 37 to it. The second statement creates `second_variable` and assigns `first_variable` to it; that is, the statement copies the value of 37 from `first_variable` into this new variable. The third statement replaces the previous value in `first_variable` with 117.

The value of 37 is copied from one variable's storage location to another, just like copying and pasting the number from B2 to C2 in the spreadsheet. Assigning 117 to `first_variable` in the third statement has no effect on the value of `second_variable`. They are two independent variables at that point.

Variable Creation

Scrolling down the spreadsheet reveals more and more empty cells. Having all these empty cells might seem like a waste of memory, but the spreadsheet doesn't actually set aside space for a cell in memory until you put some data into it.

In the same way, a Python variable is created when a value is first assigned to it. Prior to that first assignment, the variable doesn't exist in memory.

Why is this important? Consider the following program:

```
first_variable = 37
first_variable = second_variable
second_variable = 117
```

The first line of this program creates a new variable, `first_variable`, and assigns 37 to it. The second line attempts to assign the value of `second_variable` to `first_variable`, but `second_variable` hasn't been assigned a value yet, so it doesn't exist. This line will produce an error when the program is run:

```
NameError: name 'second_variable' is not defined
```

You must assign a value to a variable before referencing that variable's value.

Variable Types

Select cell B2 in the spreadsheet again and then select the option to format the cell; in Google Sheets, this is **Format ▸ Number** on the menu. The available formatting options shown demonstrate how spreadsheets can store all types of data in cells: text, whole numbers, numbers with decimal points, currency, dates, and more.

Python variables can also store many different types of data. In programming, the *type* of data indicates how it is stored and interpreted by the programming language. In this chapter, we'll focus on three fundamental types of data in Python: integers, floating-point numbers, and strings.

Integers

You've already seen the integer type in statements such as this:

```
first_variable = 37
```

The word *integer* is programming-speak for a whole number, like 37 and 117. In this assignment statement, the 37 is an integer *literal*, a specific value embedded in the text of our programming code. Each data type has rules for literals we must follow, and these rules often vary from how we would write the same value when we're not programming. One important rule for numerical literals is that you cannot include commas to separate digit groups. If you want to assign the number 4,506 to `second_variable`, you would use the literal `4506`:

```
second_variable = 4506
```

Floating-Points

A *floating-point* is a number that can represent fractional amounts. Floating-point literals have a decimal point and one or more digits after the decimal. Here is a sample line of code that creates a floating-point variable:

```
float_variable = 1245.75
```

As with integers, floating-point literals cannot include commas. Note that a numerical literal is floating-point even if the portion after the decimal point is zero:

```
also_float = 23.0
```

Strings

The *string* type stores a series of characters—a character being a letter, digit, punctuation mark, or anything else that can be typed on a keyboard. Because strings can include spaces, string literals use quotation marks to indicate where the literal begins and ends. For example:

```
string_variable = "Weekly Sales Total"
```

To be clear, the quotation marks that delimit the text are not part of the string. This statement assigns the text *Weekly Sales Total* to `string_variable`. To avoid confusion, I'll include quotes when talking about string values in the text as well. Strings can also be delimited with single quotes:

```
another_string_variable = 'Annual Sales Average:'
```

We might want to include quotation marks as part of the actual string. If a string starts with a double quote, any single quotes that appear will be part of the string, and vice versa:

```
has_single = "This isn't a problem."
has_double = 'He said, "This is no problem, either."'
```

If we want to use a double quote inside double quotes, or a single quote inside single quotes, we must use an *escape sequence*: a special sequence of characters that begins with a backslash and identifies a single character. In this case, that means putting a backslash before the quotation mark to indicate `\"` or `\'`, as shown in this example:

```
double_in_double = "This string has \"double quote marks\" but that is okay."
single_in_single = 'This string\'s okay too.'
has_both = "This string has \"many\" more \'quotes\' than is necessary."
```

For clarity, Table 3-2 shows the state of storage after these statements are executed.

Table 3-2: Variable String Values After Assignments

Variable	Value
double_in_double	This string has "double quote marks" but that is okay.
single_in_single	This string's okay too.
has_both	This string has "many" more 'quotes' than is necessary.

Escape sequences can be used to include other characters we couldn't include otherwise, such as `\n`, which is the line-feed character. The literal string `"this is\ntwo lines"` represents the text:

```
this is
two lines
```

Also, because the backslash signals an escape sequence, we have an escape sequence, `\\`, to indicate a backslash. For example, the literal string `"the program is c:\\mypython\\firstprogram.py"` represents the text:

```
the program is c:\mypython\firstprogram.py
```

Sometimes in programming we need a string with no characters in it, called a *null string*, the string equivalent of a zero. A literal null string is quotes with nothing between them:

```
null_string = ''
also_null_string = ""
```

Always remember that the rules of literals determine the type of the literal, and, if we're assigning a literal value to a variable, the type of the variable as well. If we put quotes around a number, for example, that makes a string literal:

```
string_not_integer = "23"
```

Changing Types

As with spreadsheet cells, when we assign a new value to an old variable, the type of the new value doesn't have to be the same as the old value. Values of any type can be assigned to any variable regardless of the prior contents: an integer can replace a string, a string can replace a floating-point, and so on. The following is a legal sequence of instructions in Python:

```
some_variable = 37
some_variable = 100.938
some_variable = "Now I am a string."
```

Of course, just because something is legal doesn't make it a good idea. Storing different types in the same variable might make our code hard to read. In later chapters, though, we'll see how this capability can produce flexible code that handles different types of data equally well.

BEYOND PYTHON: VARIABLE DECLARATIONS AND TYPING

Creating a variable the way we do in Python, by assigning a value to a new identifier, is known as *implicit declaration*. Some languages use *explicit declaration*, which requires a variable declaration statement before a variable can be used. For example, in C, C++, or Java, one might write:

```
int someVariable;
someVariable = 37;
```

Also, most languages require a variable to store a single type throughout the program. Here, the word `int` in the declaration makes `someVariable` an integer. Unlike in Python, we couldn't put a floating-point or string value in `someVariable` later.

Input and Output

The last features we need to cover before we can start writing useful programs are input and output. Python provides two functions for these purposes. A *function*, in programming terms, is a sequence of instructions that can be executed using the function's identifier. Later, in Chapter 8, you'll learn the details of how functions work and how to write your own functions, but along the way you'll learn several of Python's most useful built-in functions. The first of these is the aptly named `input` function. Here's an example of this function being used:

```
age = input("Please enter your age: ")
```

As you can see, this is an assignment statement. On the left we have a variable named `age`. On the right, `input` is the identifier of the function we are using. Each specific use of a function is known as a *function call*. After the function identifier we have parentheses, which enclose the function's arguments. An *argument* is data that is passed to a function for processing. Different functions can take different numbers and types of arguments, and in the case of this call to `input`, the only argument is the string literal `"Please enter your age: "`. The `input` function will display this string to the user and then wait for the user to input a response. After the user types a response and presses ENTER, whatever characters the user entered will be *returned* by the function as a string value. The returned string value will then be assigned to the variable `age`.

Note that if we don't need to give the user instructions for what to input, we can leave out the argument, but every function call requires parentheses:

```
year = input()
```

To output results, we use Python's `print` function, which displays its arguments to the Python shell. The `print` function accepts any number of arguments, and each argument can be any of the types we have covered so far. Arguments to functions are separated by commas:

input_and_print.py

```
age = input("Please enter your age: ")

print("You said your age was", age, ". Are you telling the truth?")
```

Note the blank line in this source code. Most languages, Python included, allow blank lines to be placed anywhere, and we can use them to separate different parts of a program—as we do here, separating input and output.

If the user enters 45 for `input` in the first line, the output from the `print` function will be this:

```
You said your age was 45 . Are you telling the truth?
```

Notice that when we use a variable as the argument to print, such as age, the value of the variable argument is displayed.

As you can see, there is an extra space after the 45 in the output. The print function, by default, adds a space between each item it displays. We can change this behavior by using a *named argument*, which has the form of an assignment statement but appears inside the parentheses of a function call. A named argument assigns a value to a variable used by the function's code. The named argument sep specifies what print uses as a separator between displayed items. By default, sep has a value of " " (a single space). If we want our items to run together with nothing separating them, we would assign a null string to sep:

```
print("You said your age was ", age, ". Are you telling the truth?", sep = "")
```

If you run the program with this modification, you'll see the period now comes right after the 45.

To discuss what our programs are doing, we'll adopt some shorthand verbs for input and output. In this book, when a program calls the input function, we'll say that the program *reads* a value. So we might say that the first line of *input_and_output.py* reads the user's age. When a program calls the print function, we'll say the program *outputs* or *displays* values. So if *input_and_output.py* reads 45 from the user in the first line, the last line would display or output the text shown previously.

Mathematical Expressions

In useful programs, assignment statements do more than store literal values and shuffle data from one variable to another. The real power of an assignment statement comes from an arithmetic *expression*: one or more operands combined with mathematical operators.

Here's a program that demonstrates the basic concept:

print_sum.py

```
sum = 100 + 75

print(sum)
```

Here, 100 and 75 are *operands*, and the plus sign (+) is an *operator*. In executing this statement, Python will evaluate the expression on the right, summing the integers 100 and 75 to get the integer result 175, and then assign this result to the variable sum. This program will output 175 when executed.

You can try this out yourself, entering the program into a new source code window, saving the results, and running the program as we did at the start of the chapter.

Operands can be literals, like 100 and 75 in the previous example, or variables. Consider the following program:

print_total.py

```
subtotal = 200
total = subtotal + 75
```

```
print(total)
```

The output will be 275.

Math expressions in Python can use any of the operators from algebra, although some operators use different symbols to allow us to type the expressions on a keyboard. Table 3-3 lists the Python math operators.

Table 3-3: Python Math Operators

Operator	Description	Example	Result
+	Addition	4 + 30	34
-	Subtraction	12 - 4	8
*	Multiplication	6 * 5	30
/	Division	9 / 2	4.5
//	Floor division	9 // 2	4
%	Modulo (remainder)	9 % 2	1
**	Exponent	5 ** 3	125

These operators can be combined into complex expressions, like this:

total_bill.py

```
sales_tax_rate = 0.08
items_total = 36.45 + 384.55
shipping_cost = 4.5
total_bill = items_total + items_total * sales_tax_rate + shipping_cost

print(total_bill)
```

The output of this program is 459.18. As with algebra, multiplication and division take precedence over addition and subtraction. In computing total_bill, the multiplication of items_total and sales_tax_rate occurs first, then this result (33.68) is added to items_total, and that result (454.68) is added to shipping_cost to produce the final value of 459.18, which is assigned to total_bill.

This program computes the sales tax as 8 percent of items_total and adds this amount to items_total and shipping_cost. We could get the same result by computing 108 percent of items_total and adding this amount and shipping_cost. Rewriting the code in this way requires us to use parentheses to alter the order of computation, like this:

total_bill_alternate.py

```
sales_tax_rate = 0.08
items_total = 36.45 + 384.55
shipping_cost = 4.5
total_bill = items_total * (1 + sales_tax_rate) + shipping_cost

print(total_bill)
```

With the parentheses, we add 1 to `sales_tax_rate` to make 1.08 before multiplying this result and `items_total`. Without the parentheses, the multiplication of `items_total` and 1 would happen first.

Mixing Types in Arithmetic Expressions

As we've seen, computations with integers produce integers, and computations with floating-points produce floating-points. Python also allows numerical types to be mixed in the same expression:

```
income = 35764
income_tax = 2500 + (income - 17500) * 0.15
```

When integers and floating-points appear in the same expression, the integers are converted to floating-point, and thus the final result of the calculation is floating-point as well. In the assignment of `income_tax`, the literal integer values of 2500 and 17500 and `income`'s integer value of 35764 will be converted to floating-point (2500.0, 17500.0, and 35764.0), and the result of 5239.6 will be assigned to `income_tax`.

Division is a bit of an odd duck among operators, because dividing one integer into another may produce fractional results. For example, 16 divided by 2 is 8, but 16 divided by 5 is 3.2.

In some cases, instead of fractional results, we might rather have division results expressed as a whole number plus a remainder, so that 16 divided by 5 would be expressed as 3 with a remainder of 1, rather than 3.2.

Python's answer to this issue is to provide three different operators for division. The basic `/` operator always produces a floating-point result: `16 / 5` is 3.2, and `16 / 2` is 8.0. The `//` operator always rounds the result of division *down* to the next lower number, and if both operands are integers, the result is an integer as well. This is known as *floor division*. For example, `16 // 2` is 8, `16 // 5` is 3, and `16.8 // 5` is 3.0. If the result is negative, the result is still rounded down, not rounded off; `-16 / 5` is –3.2, but the result of `-16 // 5` is –4, not –3.

BEYOND PYTHON: ROUNDING DIVISION RESULTS

Almost all programming languages distinguish between integer and floating-point division, but Python is unusual in how it handles this distinction. In Python, the programmer chooses between fractional and rounded results by choosing different division operators. In many languages, the type of a division result is determined strictly from the type of the operands, so that 9 / 4 would be integer 2 but 9.0 / 4.0 would be floating-point 2.25. Also, Python is nearly unique in choosing "floor" rounding for integer division; most languages round *off* rather than rounding *down*. This affects results with negative numbers; although –9 // 4 is –3 in Python, it will be –2 in most languages.

The third division operator is `%`, the *modulo* operator, which yields the remainder of division. So `16 % 5` is 1, and `16 % 2` is 0. Note that because `-16 // 5` is −4, as explained previously, it means the remainder, `-16 % 5`, is 4.

String Operations

Python provides many useful string operations, but we'll discuss just the most useful string operation, *concatenation*, which chains separate strings into one longer string with the `+` operator:

concatenate.py

```
big_string = "This is " + "one big" + " string now."

print(big_string)
```

This program constructs and outputs a single string:

```
"This is one big string now."
```

Sometimes numerical data will be stored as a string. In itself, this is no problem, but we can't perform calculations with a number stored as a string. For example:

string_error.py

```
string_income = "34465"
tax = string_income * 0.25

print(tax)
```

The attempt to compute the product of `string_income` and 0.25 will generate an error. In order to use a number stored in a string in an arithmetic expression, we must first convert the string to a numeric type, using either the `int` function, which returns an integer value equivalent to its string argument, or the `float` function, which returns a floating-point value. The previous example can be correctly written as follows:

int_function.py

```
string_income = "34465"
tax = int(string_income) * 0.25

print(tax)
```

This program will output `8616.25`. This situation is commonly encountered when using the `input` function, which returns whatever the user types as a string. When we are asking the user for numerical input we plan to use in an expression, we can pass the result from `input` directly to `int` or `float`, as in this example:

sales_tax.py

```
purchase_amount = 24.50
sales_tax_rate = float(input("Enter the sales tax rate as a percentage: "))
sales_tax = purchase_amount * sales_tax_rate

print(sales_tax)
```

Occasionally, a program will need to convert an integer or floating-point to a string, which is done with the str function:

tax_message.py

```
tax_amount = 3785.45
tax_message = "You owe " + str(tax_amount) + " in taxes."

print(tax_message)
```

Splitting Lines

Although most source code editors, IDLE's included, allow lines of code to be any length, we programmers tend to limit line length so that we can read an entire line without having to scroll horizontally. The official PEP 8 suggestion is to limit lines to 79 characters; as a general rule, you'll want to pick a line length based on how wide you keep the editor window on your screen.

Sometimes, though, we'll have a line that doesn't fit within our chosen width, in which case we need to split the logical statement across two or more lines. Python allows us to split a line inside a set of parentheses at any place we could put a space. In the case of computations, that means we can split a line before an operator, like this:

line_splitting.py

```
integer_variable = 500
second_variable = 200
multi_line = (34 + integer_variable
❶             + 33 + second_variable)

print(multi_line)
```

Or we can split after an operator, like this:

```
integer_variable = 500
second_variable = 200
multi_line = (34 + integer_variable +
❶             33 + second_variable)

print(multi_line)
```

The latter is my personal preference, but the former is specified by PEP 8, so that's what you will see in this book. In either case, just remember that we can only split a line inside parentheses and where we could put a space. Also note that continuation lines (marked ❶ in these two listings) are indented to line up with the first character after the left parenthesis in the previous line. We'll talk more about indentation and what it means in Python in Chapter 5. At this point, just know that if you use parentheses and press ENTER at the place where you want to split the line, the IDLE editor will indent the continuation line for you.

Now we know how to create variables, store data in them, perform calculations, read input from the user, and output results. We have everything we need to start solving simple problems.

Programs with Variables and Mathematics

Let's take another look at the sample program from the start of the chapter:

toothpaste.py

```
❶ tube_count = int(input("How many tubes to buy: "))

❷ toothpaste_cost = 1.76
❸ subtotal = toothpaste_cost * tube_count
❹ sales_tax_rate = 0.08
❺ sales_tax = subtotal * sales_tax_rate
❻ total = subtotal + sales_tax

❼ print("Toothpaste subtotal: $", subtotal, sep = "")
  print("Tax: $", sales_tax, sep = "")
  print("Total is $", total, " including tax.", sep = "")
```

This program uses only the Python shown so far in this chapter. Let's examine it line by line to see how it works.

On the first line ❶, the program requests the number of toothpaste tubes from the user using the `input` function. Because the program needs to perform arithmetic with this number later, the input string is sent to the `int` function to convert it to an integer, which is then stored in `tube_count`. Then `toothpaste_cost` is assigned the floating-point literal 1.76 ❷, the cost of one tube of toothpaste. The values of the variables `toothpaste_cost` and `tube_count` are multiplied and the result is stored in `subtotal` ❸. The floating-point literal 0.08, representing an 8 percent tax rate, is assigned to `sales_tax_rate` ❹, and the values of `subtotal` and `sales_tax_rate` are multiplied with the result assigned in `sales_tax` ❺. The sum of the pre-tax toothpaste cost, `subtotal`, and the sales tax, `sales_tax`, is assigned to `total` ❻. The program ends with three lines of calls ❼ to the `print` function, displaying the values of the `subtotal`, `sales_tax`, and `total` variables, along with some string literals containing explanatory text. Note that in each of these `print` calls, the named argument `sep` is set to the null string. Without that, each line of output would have a space between the dollar sign and the number.

In this program and others we've seen, you may have noticed that the numerical output doesn't look right for an amount of currency. If you run this program and enter 877 for the number of toothpaste tubes, the resulting total will display as $1667.0016, when we would expect to see something like $1,667.00. Python's `print` function has extra features that can help us format numbers for display, but instead we'll treat these issues as opportunities for problem solving in this and later chapters.

It's important to remember when reading code that assignment statements are value copies, the same as copying and pasting the contents of one spreadsheet cell into another. When we use the equals sign in algebra, we are establishing a relationship that is always true, as when Pythagoras tells us that for a right triangle, a2 + b2 = c2. An assignment statement, though, merely places a value in a named storage location.

The Input-Calculations-Output Program Structure

To cement your understanding of the Python language shown so far, let's take a look at another problem.

PROBLEM: METER CONVERSION

Write a program that reads a number of meters and outputs the equivalent number of centimeters and kilometers.

This is less like problem solving and more like assembling and modifying parts from a kit:

meter_conversion.py

```
❶ meters = int(input("Enter a number of meters: "))

❷ centimeters = meters * 100
❸ kilometers = meters / 1000

❹ print("That's", centimeters, "centimeters.")
  print("And", kilometers, "kilometers.")
```

You may have noticed that this program is structured much like the first one. The program calls the `input` function and passes the result to the `int` function to convert the string input to an integer. This result is stored in `meters` ❶, after which, `meters` is multiplied by 100 and the result is stored in `centimeters` ❷. Then, `meters` is divided by 1000 and the result is stored in `kilometers` ❸. Finally, the program outputs `centimeters` and `kilometers`, along with supporting text, with two calls to the `print` function ❹. This basic structure—input, calculations, and output—is at the heart of most useful programs.

Choices Programmers Make

Even in a program this short, the programmer still has choices to make. I've made `meters` and `centimeters` integer variables, but `kilometers` is a floating-point because it would potentially lose a lot of the precision if floor-division were used. I've performed all the calculations before any output; I could have interleaved the output and calculations. Or I might have placed the calculating expressions directly in the output statements, making an even shorter program:

```
meters = int(input("Enter a number of meters: "))

print("That's", meters * 100, "centimeters.")
print("And", meters / 1000, "kilometers.")
```

My choices make sense to me, but other programmers might make different choices, and that's okay. While there are lots of ways to write a bad program, there are also many ways to write a good program. Programming, like most things, involves trade-offs, and the relative importance a programmer places on various programming virtues will dictate the choices made in development. This will be a continuing theme throughout this book.

Warm-up Exercises

Like the programs shown in the previous section, this group of exercises requires no real problem solving, just direct application of the Python statements shown so far. These warm-up exercises should help you absorb the language syntax so you won't be tripping over it when we get to the trickier problems. Do not skip these exercises unless you already have experience programming in Python using `input`, `print`, and math expressions.

3-1. Write a program that reads a Fahrenheit temperature and displays the temperature in Celsius and Kelvin.

3-2. Write a program that reads an hourly wage and average number of hours worked per week and displays the total yearly pay.

3-3. Write a program that reads a total number of inches and outputs feet and inches. For example, if the input is 54, the output should be `4 feet 6 inches`.

3-4. Write a program that reads at-bats and hits and displays the batting average. (Cricket fans can compute batting average from runs and outs if they prefer.)

3-5. This one will probably require a little Googling: Write a program that computes an NCAA quarterback rating: read the relevant statistics and display the answer.

Problem Solving with Variables and Mathematics

Now we're ready to look at problems that require real problem solving. The programming itself is just as simple as the previous examples, using the same basic combination of input, calculations, and output, and the resulting programs are just as short. The difference is in the thought process that must occur before the code is written.

An important note: we'll be solving these problems using only the Python we've seen to this point, but some elements in these problems could be solved more directly using techniques we'll cover in later chapters. This pattern will repeat throughout this book. We'll try to push each language concept as far as it can go, even though there may be other ways to solve the same problem around the corner.

Why do this? For one thing, we don't want to wait until we've covered most of the language to start learning problem solving. This early exposure to problem solving will develop your problem-solving skills better than waiting. Solving problems with restricted programming syntax will allow you to fully understand the capabilities of each element of programming and help you to unlock all your creative potential as a problem solver. Exploring the limitations of simpler programming instructions also helps you understand why programming languages have the features they have.

As explained in the introduction, the Python community encourages a set of concepts they call *Pythonic* programming, and one Pythonic concept is that there is one "right" approach for each particular problem. Because of this, a solution that deliberately avoids using advanced language features may not be Pythonic. But remember that the point of this book is for you to learn to think like a programmer—to learn how to solve programming problems on your own.

Therefore, we'll write programs using the syntax we've covered to that point, and trust that our solutions will become more Pythonic as we become more knowledgeable and proficient.

Packs and Cans

Our first problem involves monetary calculations, but there's more to it than that.

PROBLEM: EFFICIENT SODA BUYING

A local store sells six-packs of soda for $3.29. Individual cans can be bought for 90 cents a can. Write a program to read the total number of soda cans desired and display how many packs should be bought to result in the lowest cost.

At first glance, this problem might seem just as straightforward as those shown previously, but it isn't. As a rule, and especially as a beginning programmer, never assume anything is trivial in programming. Always have a plan, and don't just jump into coding. If someone wanted to buy 22 cans, for example, the right number of packs and individual cans is not immediately obvious, and even less obvious is how we can produce general formulas for the answers.

Making a Table

Making a table of sample input and output is a good way to start when the right output isn't clear. Table 3-4 shows a range of input (the desired number of soda cans) from 1–12, the number of packs and individual cans that should be bought to result in the lowest cost, and that cost.

Table 3-4: Sample Input and Output for *Efficient Soda Buying*

Cans needed (input)	Six-packs (output)	Individual cans	Total cost
1	0	1	$0.90
2	0	2	$1.80
3	0	3	$2.70
4	1	0	$3.29
5	1	0	$3.29
6	1	0	$3.29

7	1	1	$4.19
8	1	2	$5.09
9	1	3	$5.99
10	2	0	$6.58
11	2	0	$6.58
12	2	0	$6.58

As you can see, buying individual cans is better when buying 1–3 cans, but then it becomes more economical to buy a six-pack, and this pattern repeats as the number of cans increases. Creating the table provides data we can use to test our program once it is written. Also, the table may give us some hints in writing our program. The output, shown in the second column of the table, has a definite pattern, but how to produce that pattern is not immediately clear.

Guessing and Testing

Because the six-pack holds six cans, it's logical to think that figuring out the number of packs we should buy will involve dividing by six. In fact, if someone wanted to buy an exact multiple of six cans, simply dividing the number of cans by six would be the right answer. That will not produce the right answer here, but let's see how wrong the result actually is.

To do that, let's augment the previous table with a new column that shows the result of dividing the total number of cans by six. Because we can't buy part of a six-pack, we'll use floor division, Python's // operator. The result is shown as Table 3-5.

Table 3-5: Floor-Division Results

Cans needed (input)	Six-packs (output)	Cans // 6
1	0	0
2	0	0
3	0	0
4	1	0
5	1	0
6	1	1
7	1	1
8	1	1
9	1	1
10	2	1
11	2	1
12	2	2

This is what I call *guessing and testing*; when you aren't sure about a mathematical formula, make an educated guess, compare the results to what you need, and see if you can bring the two together. In this case, the floor division produces the right numbers, but two rows below where we would like them. In other words, if we could just shift the third column up two rows, it would match the desired output.

A Formula for the Pattern

Maybe that's a clue to a solution. What if we used addition to shift the results? If we added 2 to the number of cans before the floor division, then, for example, an input of 4 would produce the results on row 6 of the third column. Let's apply this idea to our table to make sure it works (Table 3-6).

Table 3-6: Floor-Division Modification

Cans needed (input)	Six-packs (output)	Cans // 6	(Cans + 2) // 6
1	0	0	0
2	0	0	0
3	0	0	0
4	1	0	1
5	1	0	1
6	1	1	1
7	1	1	1
8	1	1	1
9	1	1	1
10	2	1	2
11	2	1	2
12	2	2	2

The fourth column looks just like the second—success! Now we can translate this idea into a program:

efficient_soda_buying.py

```
❶ cans_needed = int(input("Number of cans needed: "))
❷ packs = (cans_needed + 2) // 6

❸ print("You should buy", packs, "packs.")
```

Here, we've simply applied what we learned to the input-calculations-output template. Our program gets the number of cans from the user ❶, computes the number of packs using the formula we discovered ❷, and displays the results ❸.

This is another short program, but this problem offers a first glimpse of what programming and problem solving are all about. In the warm-up exercises, the calculation needed in the solution was either given in the problem or obvious. In this problem, figuring out the formula was most of the work. This is a pattern that will continue throughout this book.

Note that this problem doesn't require our program to figure out how many individual cans need to be bought. Because efficiently buying six-packs can result in buying more cans than needed, the pattern for individual cans, as shown in Table 3-4, is more complicated than the pattern for six-packs. It's possible for a program to calculate the number of individual cans, but there's a much easier way to accomplish such a task that we'll see in the next chapter.

A Better Money Display

Let's take a look at another problem that requires a different approach. In previous programs, the results of our monetary calculations have been numbers with more digits after the decimal than is customary. If a result is 7.475, for example, we'd want to display that as $7.48. How can we do that using just the tools shown so far?

PROBLEM: ROUNDING MONEY

Write a program that reads an arbitrary floating-point number and displays it with exactly two decimal places.

Many new programmers would have no idea how to get started on this problem, because it doesn't seem to have anything to do with the Python we've learned so far. Our toolbox is filled only with tools for calculations, but no calculation appears to be involved here.

This problem is really a puzzle as much as a program, and to solve it, we need to apply the lesson from the fox, goose, and corn puzzle. Instead of thinking about what needs to be done, we should think about the operations taught in this chapter and what they can be made to do, and work from there. We know that floor division rounds its result down. It's a tool that is typically used with integers, but we can experiment to see what we can make it do. What happens if we floor-divide a floating-point by 1?

divide_by_one.py

```
  float_num = 365.4386
❶ float_num = float_num // 1

  print(float_num)
```

This code is straightforward, but I do want to mention the second statement, where `float_num` is used on both sides of an assignment statement ❶. As an algebraic expression, this statement wouldn't make any sense, but it is perfectly legal and happens all the time in programming. The statement `float_num = float_num // 1` takes the value currently in `float_num`, floor-divides it by 1, and stores the result back in `float_num`, overwriting whatever was there before.

Note that, as with previous programs, the calculations could be performed in a single assignment statement, but using separate statements makes the code more readable and matches our thought process: first, we have a number with four decimal places, and then we try to round it off.

And our effort proves partially successful, because when we run this program, the output is 365.0. This method will round off a floating-point number—but to a whole number; the number has been rounded to 365 but displays as 365.0 because it's a floating-point. We ask ourselves if there's any way to move the decimal point, and there is; we can just multiply or divide by powers of 10. We extend the experiment, and multiply `float_num` by 100 before floor-dividing by 1, then divide by 100 afterward.

two_decimal_places.py

```
float_num = 365.4386
float_num = float_num * 100
float_num = float_num // 1
float_num = float_num / 100

print(float_num)
```

Running this code, we get a result of 365.43, rounded to two decimal places—time to pop the cork on the champagne! But when we look more closely, we see that we started with 365.4386, and since we should round our decimals to the nearest number and not just round down, this should round to 365.44.

Floor division always rounds down for positive numbers, but maybe there is still some way to get floor division to do what we want. So we think back to the *Efficient Soda Buying* problem and how we used addition to shift the result of floor division to the result we wanted. Is there a way to do the same here?

Consider the number 1.01. We want any floating-point number from 1.005 to just under 1.015 to round to 1.01. With our current code, though, the range of numbers that round to 1.01 is 1.01 to just under 1.02. These ranges are 0.005 apart. If we add 0.005 to a number before rounding, we will bring it into the range that will round down to the number we want.

rounding_money.py

```
float_num = 365.4386
float_num = float_num + 0.005
float_num = float_num * 100
float_num = float_num // 1
float_num = float_num / 100

print(float_num)
```

We run this program and the results look correct. Now we want to test this on a variety of numbers, so now is a good time to change the literal floating-point in the first statement to a call to `input`:

```
float_num = float(input("Enter a floating-point number: "))
```

Starting with a literal value in the assignment makes it easier to test when we were using the same test value for every run. Once we're closer to a working solution, though, having the program read the input is easier than modifying the code every time we want to change the test data. Not every programmer would do this. This is the kind of choice that marks a programmer's individual personality. By the end of this book, you'll be putting together your own personal plan for program development.

When we test this code with different inputs, the results all look properly rounded. The results aren't always perfect, though. If we enter 245.1023, for example, the output is 245.1, when we want it to be 245.10. How can we force the second digit after the decimal to display even when it is zero? That problem can be solved using only the tools of this chapter, but I'm leaving the solution up to you.

A First Taste of Problem Solving

By now you can see that solving problems and learning Python are two different things, and it's in learning both that you will become a programmer. Notice how I used some of the problem-solving techniques from the classic puzzles in the first chapter to work out programming solutions. In the rest of this book, you will learn a lot more about the Python language and programming in general, but you'll use the techniques in this chapter again and again. Eventually, their use will become second nature.

Problem-Solving Exercises

Now it's your turn. Earlier in the chapter you worked through some warm-up exercises, but these are the first problem-solving exercises in the book. Because the ideas in this chapter are the basis for everything that follows, it's important that you don't go ahead until you've completed these exercises. Work through the problems in the order shown. Remember to apply the problem-solving techniques shown in this chapter. Don't tackle the full problems head on; use tables to visualize patterns of results.

One further note: some readers will have already learned more Python than what is covered in this chapter, but don't be Captain Kirk: make sure you only use the basic tools of this chapter in solving these problems.

3-6. A rock band needs to book studio time to record their new album. The studio books time in half-hour increments at $120 per half-hour. Write a program that allows the musicians to enter the total time they need to record their album in minutes, and outputs the studio cost. For example, if they needed 415 minutes, that would require 14 half-hour sessions, at a total cost of $1,680.

3-7. A store sells soda by the case. A case holds 24 bottles and cost $22.75. Write a program that allows the user to enter the minimum number of desired bottles, and outputs the number of bottles that will actually have to be purchased as well as the cost. For example, if the user enters 80, the program will output that 96 bottles will be purchased at a cost of $91.

3-8. The rules for determining a leap year are as follows: a year is a leap year if the year number is divisible by 4, except that years divisible by 100 are not leap years unless they are also divisible by 400. Thus, the year 1900 was not a leap year, but the year 2000 was. Write a program that asks for a number of years and determines how many of those years, starting at the year 2001, are leap years. That is, if the user enters 102, the program will determine how many years in the range 2001–2103 are leap years (the answer is 19). This problem may be too much to try in one pass. Consider starting with the basic rule (every fourth year is a leap year), and adding the exceptions once the program produces the right answer for the basic rule.

3-9. Write a program that reads a three-digit number and displays the sum of the three digits. For example, if the input is 429, the output should be 15.

3-10. You knew this was coming: rewrite the Rounding Money code so that it always displays two decimal places. For example, an input of 1.001 should produce an output of 1.00.