Chapter 6

Managing Information

In This Chapter

- Understanding the Python view of data
- Using operators to assign, modify, and compare data
- Organizing code using functions
- Interacting with the user

Whether you use the term *information* or *data* to refer to the content that applications manage, the fact is that you must provide some means of working with it or your application really doesn't have a purpose. Throughout the rest of the book, you see *information* and *data* used interchangeably because they really are the same thing, and in real-world situations, you'll encounter them both, so getting used to both is a good idea. No matter which term you use, you need some means of assigning data to variables, modifying the content of those variables to achieve specific goals, and comparing the result you receive with desired results. This chapter addresses all three requirements so that you can start to control data within your applications.

It's also essential to start working through methods of keeping your code understandable. Yes, you could write your application as a really long procedure, but trying to understand such a procedure is incredibly hard, and you'd find yourself repeating some steps because they must be done more than once. Functions are one way for you to package code so that it's easier to understand and reuse as needed.

Applications also need to interact with the user. Yes, some perfectly usable applications are out there that don't really interact with the user, but they're extremely rare and don't really do much, for the most part. In order to provide a useful service, most applications interact with the user to discover how the user wants to manage data. You get an overview of this process in this chapter. Of course, you visit the topic of user interaction quite often throughout the book because it's an important topic.

Controlling How Python Views Data

As discussed in Chapter 5, all data on your computer is stored as 0s and 1s. The computer doesn't understand the concept of letters, Boolean values, dates, times, or any other kind of information except numbers. In addition, a computer's capability to work with numbers is both inflexible and relatively simplistic. When you work with a string in Python, you're depending on Python to translate the concept of a string into a form the computer can understand. The storage containers that your application creates and uses in the form of variables tell Python how to treat the 0s and 1s that the computer has stored. So, it's important to understand that the Python view of data isn't the same as your view of data or the computer's view of data — Python acts as an intermediary to make your applications functional.

To manage data within an application, the application must control the way in which Python views the data. The use of operators, packaging methods such as functions, and the introduction of user input all help applications control data. All these techniques rely, in part, on making comparisons. Determining what to do next means understanding what state the data is in now as compared to some other state. If the variable contains the name John now, but you really want it to contain Mary instead, then you first need to know that it does in fact contain John. Only then can you make the decision to change the content of the variable to Mary.

Making comparisons

Python's main method for making comparisons is through the use of operators. In fact, operators play a major role in manipulating data as well. The upcoming "Working with Operators" section discusses how operators work and how you can use them in applications to control data in various ways. Later chapters use operators extensively as you discover techniques for creating applications that can make decisions, perform tasks repetitively, and interact with the user in interesting ways. However, the basic idea behind operators is that they help applications perform various types of comparisons.

In some cases, you use some fancy methods for performing comparisons in an application. For example, you can compare the output of two functions (as described in the "Comparing function output" section, later in this chapter). With Python, you can perform comparisons at a number of levels so that you can manage data without a problem in your application. Using these techniques hides detail so that you can focus on the point of the comparison and define how to react to that comparison rather than become mired in detail.

Your choice of techniques for performing comparisons affects the manner in which Python views the data and determines the sorts of things you can do to manage the data after the comparison is made. All this functionality might seem absurdly complex at the moment, but the important point to remember is that applications require comparisons in order to interact with data correctly.

Understanding how computers make comparisons

Computers don't understand packaging, such as functions, or any of the other structures that you create with Python. All this packaging is for your benefit, not the computer's. However, computers do directly support the concept of operators. Most Python operators have a direct corollary with a command that the computer understands directly. For example, when you ask whether one number is greater than another number, the computer can actually perform this computation directly, using an operator. (The upcoming section explains operators in detail.)

Some comparisons aren't direct. Computers work only with numbers. So, when you ask Python to compare two strings, what Python actually does is compare the numeric value of each character in the string. For example, the letter *A* is actually the number 65 in the computer. A lowercase letter *a* has a different numeric value — 97. As a result, even though you might see ABC as being equal to abc, the computer doesn't agree — it sees them as different because the numeric values of their individual letters are different.

Working with Operators

Operators are the basis for both control and management of data within applications. You use operators to define how one piece of data is compared to another and to modify the information within a single variable. In fact, operators are essential to performing any sort of math-related task and to assigning data to variables in the first place.

When using an operator, you must supply either a variable or an expression. You already know that a variable is a kind of storage box used to hold data. An *expression* is an equation or formula that provides a description of a mathematical concept. In most cases, the result of evaluating an expression is a Boolean (true or false) value. The following sections describe operators in detail because you use them everywhere throughout the rest of the book.

Understanding Python's one ternary operator

A ternary operator requires three elements. Python supports just one such operator, and you use it to determine the truth value of an expression. This operator takes the following form:

```
TrueValue if Expression else
   FalseValue
```

When the `Expression` is true, the operator outputs `TrueValue`. When the expression is false, it outputs `FalseValue`. As an example, if you type

```
"Hello" if True else
   "Goodbye"
```

the operator outputs a response of `'Hello'`. However, if you type

```
"Hello" if False else
   "Goodbye"
```

the operator outputs a response of `'Goodbye'`. This is a handy operator for times when you need to make a quick decision and don't want to write a lot of code to do it.

One of the advantages of using Python is that it normally has more than one way to do things. Python has an alternative form of this ternary operator — an even shorter shortcut. It takes the following form:

```
(FalseValue, TrueValue)
   [Expression]
```

As before, when `Expression` is true, the operator outputs `TrueValue`; otherwise, it outputs `FalseValue`. Notice that the `TrueValue` and `FalseValue` elements are reversed in this case. An example of this version is

```
("Hello", "Goodbye")[True]
```

In this case, the output of the operator is `'Goodbye'` because that's the value in the `TrueValue` position. Of the two forms, the first is a little clearer, while the second is shorter.

Defining the operators

An *operator* accepts one or more inputs in the form of variables or expressions, performs a task (such as comparison or addition), and then provides an output consistent with that task. Operators are classified partially by their effect and partially by the number of elements they require. For example, a unary operator works with a single variable or expression; a binary operator requires two.

The elements provided as input to an operator are called *operands.* The operand on the left side of the operator is called the left operand, while the operand on the right side of the operator is called the right operand. The following list shows the categories of operators that you use within Python:

- Unary
- Arithmetic
- Relational

- Logical
- Bitwise
- Assignment
- Membership
- Identity

Each of these categories performs a specific task. For example, the arithmetic operators perform math-based tasks, while relational operators perform comparisons. The following sections describe the operators based on the category in which they appear.

Unary

Unary operators require a single variable or expression as input. You often use these operators as part of a decision-making process. For example, you might want to find something that isn't like something else. Table 6-1 shows the unary operators.

Table 6-1 Python Unary Operators

Operator	*Description*	*Example*
~	Inverts the bits in a number so that all the 0 bits become 1 bits and vice versa.	~4 results in a value of –5
-	Negates the original value so that positive becomes negative and vice versa.	–(–4) results in 4 and –4 results in –4
+	Is provided purely for the sake of completeness. This operator returns the same value that you provide as input.	+4 results in a value of 4

Arithmetic

Computers are known for their capability to perform complex math. However, the complex tasks that computers perform are often based on much simpler math tasks, such as addition. Python provides access to libraries that help you perform complex math tasks, but you can always create your own libraries of math functions using the simple operators found in Table 6-2.

Table 6-2 Python Arithmetic Operators

Operator	*Description*	*Example*
+	Adds two values together	5 + 2 = 7
-	Subtracts the right operand from the left operand	5 – 2 = 3
*	Multiplies the right operand by the left operand	5 * 2 = 10
/	Divides the left operand by the right operand	5 / 2 = 2.5
%	Divides the left operand by the right operand and returns the remainder	5 % 2 = 1
**	Calculates the exponential value of the right operand by the left operand	5 ** 2 = 25
//	Performs integer division, in which the left operand is divided by the right operand and only the whole number is returned (also called floor division)	5 // 2 = 2

Relational

The relational operators compare one value to another and tell you when the relationship you've provided is true. For example, 1 is less than 2, but 1 is never greater than 2. The truth value of relations is often used to make decisions in your applications to ensure that the condition for performing a specific task is met. Table 6-3 describes the relational operators.

Table 6-3 Python Relational Operators

Operator	*Description*	*Example*
==	Determines whether two values are equal. Notice that the relational operator uses two equals signs. A mistake many developers make is using just one equals sign, which results in one value being assigned to another.	1 == 2 is False
!=	Determines whether two values are not equal. Some older versions of Python allowed you to use the <> operator in place of the != operator. Using the <> operator results in an error in current versions of Python.	1 != 2 is True
>	Verifies that the left operand value is greater than the right operand value.	1 > 2 is False

Operator	*Description*	*Example*
<	Verifies that the left operand value is less than the right operand value.	1 < 2 is True
>=	Verifies that the left operand value is greater than or equal to the right operand value.	1 >= 2 is False
<=	Verifies that the left operand value is less than or equal to the right operand value.	1 <= 2 is True

Logical

The logical operators combine the true or false value of variables or expressions so that you can determine their resultant truth value. You use the logical operators to create Boolean expressions that help determine whether to perform tasks. Table 6-4 describes the logical operators.

Table 6-4 Python Logical Operators

Operator	*Description*	*Example*
and	Determines whether both operands are true.	True and True is True
		True and False is False
		False and True is False
		False and False is False
or	Determines when one of two operands is true.	True or True is True
		True or False is True
		False or True is True
		False or False is False
not	Negates the truth value of a single operand. A true value becomes false and a false value becomes true.	not True is False
		not False is True

Bitwise

The bitwise operators interact with the individual bits in a number. For example, the number 6 is actually 0b0110 in binary.

If your binary is a little rusty, you can use the handy Binary to Decimal to Hexadecimal Converter at `http://www.mathsisfun.com/binary-decimal-hexadecimal-converter.html`. You need to enable JavaScript to make the site work.

A bitwise operator would interact with each bit within the number in a specific way. When working with a logical bitwise operator, a value of 0 counts as false and a value of 1 counts as true. Table 6-5 describes the bitwise operators.

Table 6-5 Python Bitwise Operators

Operator	*Description*	*Example*
`&` (And)	Determines whether both individual bits within two operators are true and sets the resulting bit to true when they are.	0b1100 & 0b0110 = 0b0100
`|` (Or)	Determines whether either of the individual bits within two operators is true and sets the resulting bit to true when one of them is.	0b1100 | 0b0110 = 0b1110
`^` (Exclusive or)	Determines whether just one of the individual bits within two operators is true and sets the resulting bit to true when one is. When both bits are true or both bits are false, the result is false.	0b1100 ^ 0b0110 = 0b1010
`~` (One's complement)	Calculates the one's complement value of a number.	~0b1100 = –0b1101 ~0b0110 = –0b0111
`<<` (Left shift)	Shifts the bits in the left operand left by the value of the right operand. All new bits are set to 0 and all bits that flow off the end are lost.	0b00110011 << 2 = 0b11001100
`>>` (Right shift)	Shifts the bits in the left operand right by the value of the right operand. All new bits are set to 0 and all bits that flow off the end are lost.	0b00110011 >> 2 = 0b00001100

Assignment

The assignment operators place data within a variable. The simple assignment operator appears in previous chapters of the book, but Python offers a number of other interesting assignment operators that you can use. These other assignment operators can perform mathematical tasks during the assignment process, which makes it possible to combine assignment with a math operation. Table 6-6 describes the assignment operators. For this particular table, the initial value of MyVar in the Example column is 5.

Table 6-6 Python Assignment Operators

Operator	*Description*	*Example*
=	Assigns the value found in the right operand to the left operand.	MyVar = 2 results in MyVar containing 2
+=	Adds the value found in the right operand to the value found in the left operand and places the result in the left operand.	MyVar += 2 results in MyVar containing 7
-=	Subtracts the value found in the right operand from the value found in the left operand and places the result in the left operand.	MyVar -= 2 results in MyVar containing 3
*=	Multiplies the value found in the right operand by the value found in the left operand and places the result in the left operand.	MyVar *= 2 results in MyVar containing 10
/=	Divides the value found in the left operand by the value found in the right operand and places the result in the left operand.	MyVar /= 2 results in MyVar containing 2.5
%=	Divides the value found in the left operand by the value found in the right operand and places the remainder in the left operand.	MyVar %= 2 results in MyVar containing 1
**=	Determines the exponential value found in the left operand when raised to the power of the value found in the right operand and places the result in the left operand.	MyVar **= 2 results in MyVar containing 25
//=	Divides the value found in the left operand by the value found in the right operand and places the integer (whole number) result in the left operand.	MyVar //= 2 results in MyVar containing 2

Membership

The membership operators detect the appearance of a value within a list or sequence and then output the truth value of that appearance. Think of the membership operators as you would a search routine for a database. You enter a value that you think should appear in the database, and the search routine finds it for you or reports that the value doesn't exist in the database. Table 6-7 describes the membership operators.

Table 6-7 Python Membership Operators

Operator	*Description*	*Example*
`In`	Determines whether the value in the left operand appears in the sequence found in the right operand.	"Hello" in "Hello Goodbye" is True
`not in`	Determines whether the value in the left operand is missing from the sequence found in the right operand.	"Hello" not in "Hello Goodbye" is False

Identity

The identity operators determine whether a value or expression is of a certain class or type. You use identity operators to ensure that you're actually working with the sort of information that you think you are. Using the identity operators can help you avoid errors in your application or determine the sort of processing a value requires. Table 6-8 describes the identity operators.

Table 6-8 Python Identity Operators

Operator	*Description*	*Example*
`Is`	Evaluates to true when the type of the value or expression in the right operand points to the same type in the left operand.	type(2) is int is True
`is not`	Evaluates to true when the type of the value or expression in the right operand points to a different type than the value or expression in the left operand.	type(2) is not int is False

Understanding operator precedence

When you create simple statements that contain just one operator, the order of determining the output of that operator is also simple. However, when you start working with multiple operators, it becomes necessary to determine which operator to evaluate first. For example, it's important to know whether 1 + 2 * 3 evaluates to 7 (where the multiplication is done first) or 9 (where the addition is done first). An order of operator precedence tells you that the answer is 7 unless you use parentheses to override the default order. In this case, (1 + 2) * 3 would evaluate to 9 because the parentheses have a higher order of precedence than multiplication does. Table 6-9 defines the order of operator precedence for Python.

Table 6-9 Python Operator Precedence

Operator	*Description*
`()`	You use parentheses to group expressions and to override the default precedence so that you can force an operation of lower precedence (such as addition) to take precedence over an operation of higher precedence (such as multiplication).
`**`	Exponentiation raises the value of the left operand to the power of the right operand.
`~ + -`	Unary operators interact with a single variable or expression.
`* / % //`	Multiply, divide, modulo, and floor division.
`+ -`	Addition and subtraction.
`>> <<`	Right and left bitwise shift.
`&`	Bitwise AND.
`^ |`	Bitwise exclusive OR and standard OR.
`<= < > >=`	Comparison operators.
`== !=`	Equality operators.
`= %= /= //= -= += *= **=`	Assignment operators.
`Is` `is not`	Identity operators.
`In` `not in`	Membership operators.
`not or and`	Logical operators.

Creating and Using Functions

To manage information properly, you need to organize the tools used to perform the required tasks. Each line of code that you create performs a specific task, and you combine these lines of code to achieve a desired result. Sometimes you need to repeat the instructions with different data, and in some cases your code becomes so long that it's hard to keep track of what each part does. Functions serve as organization tools that keep your code neat and tidy. In addition, functions make it easy to reuse the instructions you've created as needed with different data. This section of the chapter tells you all about functions. More important, in this section you start creating your first serious applications in the same way that professional developers do.

Viewing functions as code packages

You go to your closet, open the door, and everything spills out. In fact, it's an avalanche, and you're lucky that you've survived. That bowling ball in the top shelf could have done some severe damage! However, you're armed with storage boxes and soon you have everything in the closet in neatly organized boxes. The shoes go in one box, games in another, and old cards and letters in yet another. After you're done, you can find anything you want in the closet without fear of injury. Functions are just like that — they take messy code and place it in packages that make it easy to see what you have and understand how it works.

Commentaries abound on just what functions are and why they're necessary, but when you boil down all that text, it comes down to a single idea: Functions provide a means of packaging code to make it easy to find and access. If you can think of functions as organizers, you find that working with them is much easier. For example, you can avoid the problem that many developers have of stuffing the wrong items in a function. All your functions will have a single purpose, just like those storage boxes in the closet.

Understanding code reusability

You go to your closet, take out pants and shirt, remove the labels, and put them on. At the end of the day, you take everything off and throw it in the trash. Hmmm . . . That really isn't what most people do. Most people take the clothes off, wash them, and then put them back into the closet for reuse. Functions are reusable, too. No one wants to keep repeating the same task; it becomes monotonous and boring. When you create a function,

you define a package of code that you can use over and over to perform the same task. All you need to do is tell the computer to perform a specific task by telling it which function to use. The computer faithfully executes each instruction in the function absolutely every time you ask it to do so.

When you work with functions, the code that needs services from the function is named the *caller,* and it calls upon the function to perform tasks for it. Much of the information you see about functions refers to the caller. The caller must supply information to the function, and the function returns information to the caller.

At one time, computer programs didn't include the concept of code reusability. As a result, developers had to keep reinventing the same code. It didn't take long for someone to come up with the idea of functions, though, and the concept has evolved over the years until functions have become quite flexible. You can make functions do anything you want. Code reusability is a necessary part of applications to

- Reduce development time
- Reduce programmer error
- Increase application reliability
- Allow entire groups to benefit from the work of one programmer
- Make code easier to understand
- Improve application efficiency

In fact, functions do a whole list of things for applications in the form of reusability. As you work through the examples in this book, you see how reusability makes your life significantly easier. If not for reusability, you'd still be programming by plugging 0s and 1s into the computer by hand.

Defining a function

Creating a function doesn't require much work. Python tends to make things fast and easy for you. The following steps show you the process of creating a function that you can later access:

1. **Open a Python Shell window.**

   You see the familiar Python prompt.

2. **Type** def Hello(): **and press Enter.**

This step tells Python to define a function named Hello. The parentheses are important because they define any requirements for using the function. (There aren't any requirements in this case.) The colon at the end tells Python that you're done defining the way in which people will access the function. Notice that the insertion pointer is now indented, as shown in Figure 6-1. This indentation is a reminder that you must give the function a task to perform.

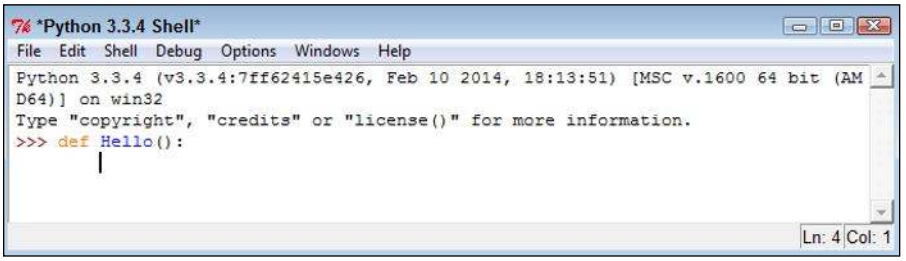

Figure 6-1: Define the name of your function.

3. **Type** print("This is my first Python function!") **and press Enter.**

   You should notice two things, as shown in Figure 6-2. First, the insertion pointer is still indented because IDLE is waiting for you to provide the next step in the function. Second, Python hasn't executed the `print()` function because it's part of a function and is not in the main part of the window.

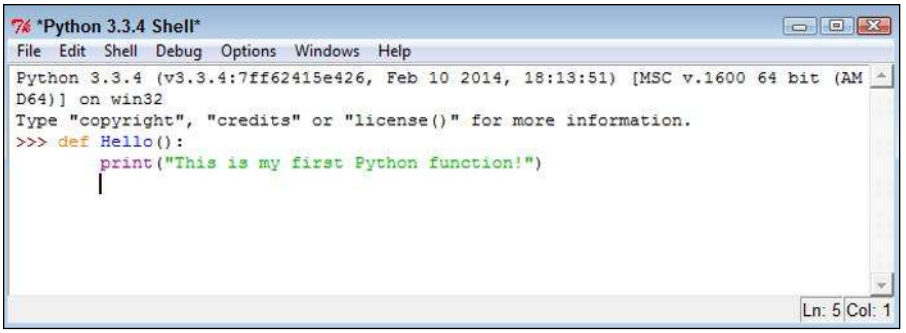

Figure 6-2: IDLE is waiting for your next instruction.

4. **Press Enter.**

   The function is now complete. You can tell because the insertion point is now to the left side, as shown in Figure 6-3. In addition, the Python prompt (>>>) has returned.

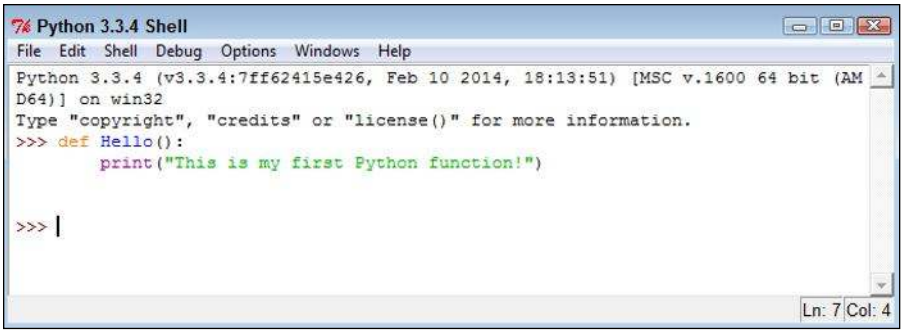

Figure 6-3: The function is complete, and IDLE waits for you to provide another instruction.

Even though this is a really simple function, it demonstrates the pattern you use when creating any Python function. You define a name, provide any requirements for using the function (none in this case), and provide a series of steps for using the function. A function ends when an extra line is added (you press Enter twice).

Working with functions in the Edit window is the same as working with them in the Python Shell window, except that you can save the Edit window content to disk. This example also appears with the downloadable source code as `FirstFunction.py`. Try loading the file into an Edit window using the same technique you use in the "Using the Edit window" section of Chapter 4.

Accessing functions

After you define a function, you probably want to use it to perform useful work. Of course, this means knowing how to access the function. In the previous section, you create a new function named `Hello()`. To access this function, you type **Hello()** and press Enter. Figure 6-4 shows the output you see when you execute this function.

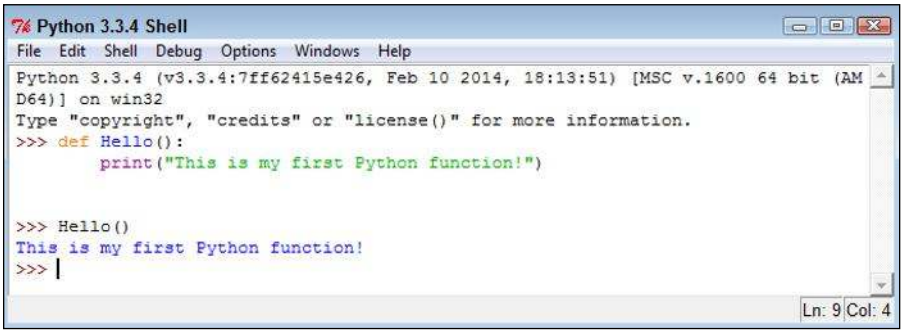

Figure 6-4: Whenever you type the function's name, you get the output the function provides.

Every function you create will provide a similar pattern of usage. You type the function name, an open parenthesis, any required input, and a close parenthesis; then you press Enter. In this case, you have no input, so all you type is **Hello()**. As the chapter progresses, you see other examples for which input is required.

Sending information to functions

The `FirstFunction.py` example is nice because you don't have to keep typing that long string every time you want to say `Hello()`. However, it's also quite limited because you can use it to say only one thing. Functions should be flexible and allow you to do more than just one thing. Otherwise, you end up writing a lot of functions that vary by the data they use rather than the functionality they provide. Using arguments helps you create functions that are flexible and can use a wealth of data.

Understanding arguments

The term *argument* doesn't mean that you're going to have a fight with the function; it means that you supply information to the function to use in processing a request. Perhaps a better word for it would be input, but the term *input* has been used for so many other purposes that developers decided to use something a bit different: argument. Although the purpose of an argument might not be clear from its name, understanding what it does is relatively straightforward. An argument makes it possible for you to send data to the function so that the function can use it when performing a task. Using arguments makes your function more flexible.

The `Hello()` function is currently inflexible because it prints just one string. Adding an argument to the function can make it a lot more flexible because you can send strings to the function to say anything you want. To see how arguments work, create a new function in the Python Shell window (or open the `Arguments01.py` file of the downloadable source; see the Introduction for the URL). This version of `Hello()`, `Hello2()`, requires an argument:

```
def Hello2( Greeting ):
    print(Greeting)
```

Notice that the parentheses are no longer empty. They contain a word, `Greeting`, which is the argument for `Hello2()`. The `Greeting` argument is actually a variable that you can pass to `print()` in order to see it onscreen.

Sending required arguments

You have a new function, `Hello2()`. This function requires that you provide an argument to use it. At least, that's what you've heard so far. Type **Hello2()** and press Enter in the Python Shell window. You see an error message, as shown in Figure 6-5, telling you that `Hello2()` requires an argument.

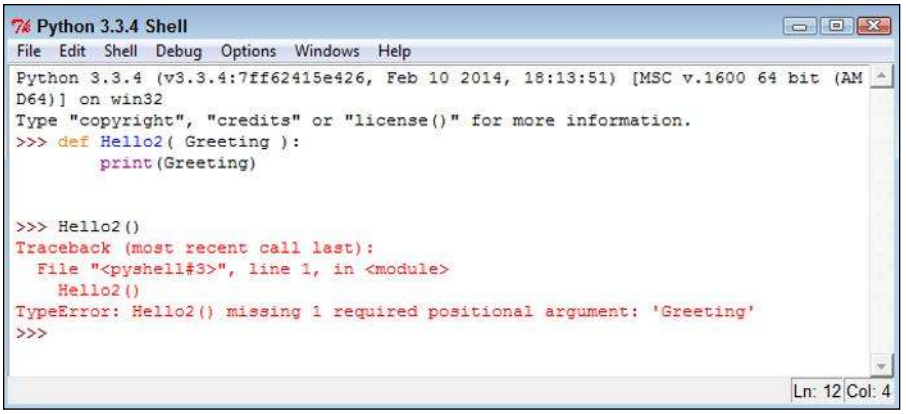

Figure 6-5: You must supply an argument or you get an error message.

Not only does Python tell you that the argument is missing, it tells you the name of the argument as well. Creating a function the way you have done so far means that you must supply an argument. Type **Hello2("This is an interesting function.")** and press Enter. This time, you see the expected output. However, you still don't know whether `Hello2()` is flexible enough to print multiple messages. Type **Hello2("Another message...")** and press Enter. You see the expected output again, as shown in Figure 6-6, so `Hello2()` is indeed an improvement over `Hello()`.

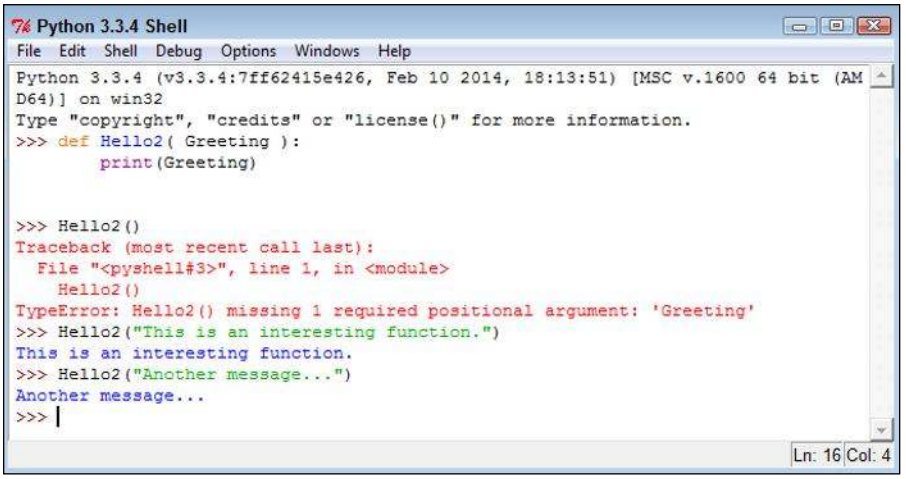

Figure 6-6: Use `Hello2()` to print any message you desire.

You might easily to assume that `Greeting` will accept only a string from the tests you have performed so far. **Type Hello2(1234)**, press Enter, and you see 1234 as the output. Likewise, type **Hello2(5 + 5)** and press Enter. This time you see the result of the expression, which is 10.

Sending arguments by keyword

As your functions become more complex and the methods to use them do as well, you may want to provide a little more control over precisely how you call the function and provide arguments to it. Up until now, you have *positional arguments,* which means that you have supplied values in the order in which they appear in the argument list for the function definition. However, Python also has a method for sending arguments by keyword. In this case, you supply the name of the argument followed by an equals sign (=) and the argument value. To see how this works, open a Python Shell window and type the following function (which is also found in the `Arguments02.py` file):

```
def AddIt(Value1, Value2):
    print(Value1, " + ", Value2, " = ", (Value1 + Value2))
```

Notice that the `print()` function argument includes a list of items to print and that those items are separated by commas. In addition, the arguments are of different types. Python makes it easy to mix and match arguments in this manner.

Time to test `AddIt()`. Of course, you want to try the function using positional arguments first, so type **AddIt(2, 3)** and press Enter. You see the expected output of `2 + 3 = 5`. Now type **AddIt(Value2 = 3, Value1 = 2)** and press Enter. Again, you receive the output `2 + 3 = 5` even though the position of the arguments has been reversed.

Giving function arguments a default value

Whether you make the call using positional arguments or keyword arguments, the functions to this point have required that you supply a value. Sometimes a function can use default values when a common value is available. Default values make the function easier to use and less likely to cause errors when a developer doesn't provide an input. To create a default value, you simply follow the argument name with an equals sign and the default value. To see how this works, open a Python Shell window and type the following function (which you can also find in the `Arguments03.py` file):

```
def Hello3(Greeting = "No Value Supplied"):
    print(Greeting)
```

This is yet another version of the original `Hello()` and updated `Hello2()` functions, but `Hello3()` automatically compensates for individuals who don't supply a value. When someone tries to call `Hello3()` without an argument, it doesn't raise an error. Type **Hello3()** and press Enter to see for yourself. Type **Hello3("This is a string.")** to see a normal response. Lest you think the function is now unable to use other kinds of data, type **Hello3(5)** and press Enter; then **Hello3(2 + 7)** and press Enter. Figure 6-7 shows the output from all these tests.

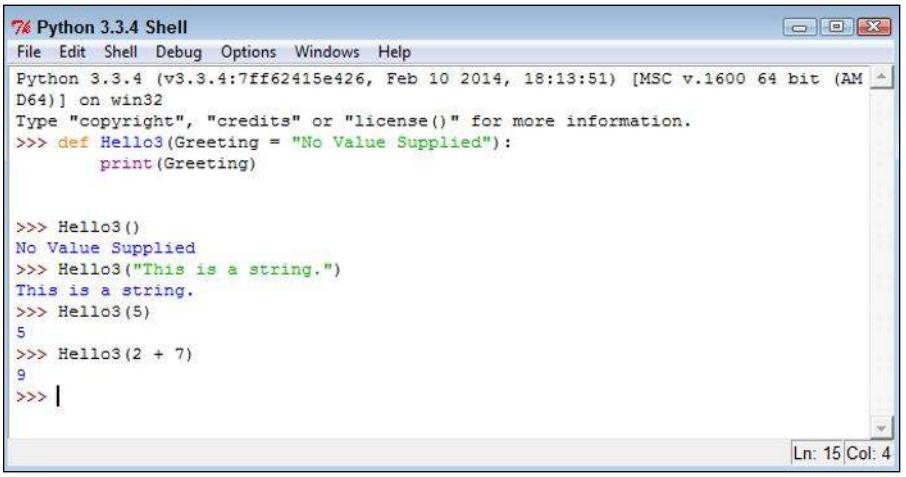

Figure 6-7: Supply default arguments when possible to make your functions easier to use.

Creating functions with a variable number of arguments

In most cases, you know precisely how many arguments to provide with your function. It pays to work toward this goal whenever you can because functions with a fixed number of arguments are easier to troubleshoot later. However, sometimes you simply can't determine how many arguments the function will receive at the outset. For example, when you create a Python application that works at the command line, the user might provide no arguments, the maximum number of arguments (assuming there is one), or any number of arguments in between.

Fortunately, Python provides a technique for sending a variable number of arguments to a function. You simply create an argument that has an asterisk in front of it, such as `*VarArgs`. The usual technique is to provide a second argument that contains the number of arguments passed as an input. Here is an example (also found in the `VarArgs.py` file) of a function that can print a variable number of elements. (Don't worry too much if you don't understand it completely now — you haven't seen some of these techniques used before.)

```
def Hello4(ArgCount, *VarArgs):
    print("You passed ", ArgCount, " arguments.")
    for Arg in VarArgs:
       print(Arg)
```

This example uses something called a `for` loop. You meet this structure in Chapter 8. For now, all you really need to know is that it takes the arguments out of `VarArgs` one at a time, places the individual argument into `Arg`, and then prints `Arg` using `print()`. What should interest you most is seeing how a variable number of arguments can work.

After you type the function into a new Python Shell window, type **Hello4(1, "A Test String.")** and press Enter. You should see the number of arguments and the test string as output — nothing too exiting there. However, now type **Hello4(3, "One", "Two", "Three")** and press Enter. As shown in Figure 6-8, the function handles the variable number of arguments without any problem at all.

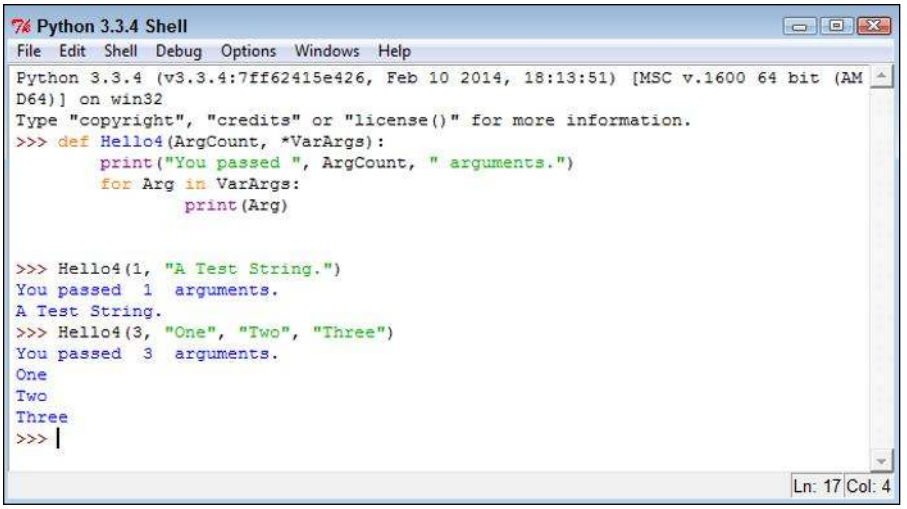

Figure 6-8: Variable argument functions can make your applications more flexible.

Returning information from functions

Functions can display data directly or they can return the data to the caller so that the caller can do something more with it. In some cases, a function displays data directly as well as returns data to the caller, but it's more common for a function to either display the data directly or to return it to the caller.

Just how functions work depends on the kind of task the function is supposed to perform. For example, a function that performs a math-related task is more likely to return the data to the caller than certain other functions.

To return data to a caller, a function needs to include the keyword `return`, followed by the data to return. You have no limit on what you can return to a caller. Here are some types of data that you commonly see returned by a function to a caller:

- **Values:** Any value is acceptable. You can return numbers, such as 1 or 2.5; strings, such as "Hello There!"; or Boolean values, such as True or False.
- **Variables:** The content of any variable works just as well as a direct value. The caller receives whatever data is stored in the variable.
- **Expressions:** Many developers use expressions as a shortcut. For example, you can simply return A + B rather than perform the calculation, place the result in a variable, and then return the variable to the caller. Using the expression is faster and accomplishes the same task.
- **Results from other functions:** You can actually return data from another function as part of the return of your function.

It's time to see how return values work. Open a Python Shell window and type the following code (or open the `ReturnValue.py` file instead):

```
def DoAdd(Value1, Value2):
    return Value1 + Value2
```

This function accepts two values as input and then returns the sum of those two values. Yes, you could probably perform this task without using a function, but this is how many functions start. To test this function, type **print("The sum of 3 + 4 is ", DoAdd(3, 4))** and press Enter. You see the expected output shown in Figure 6-9.

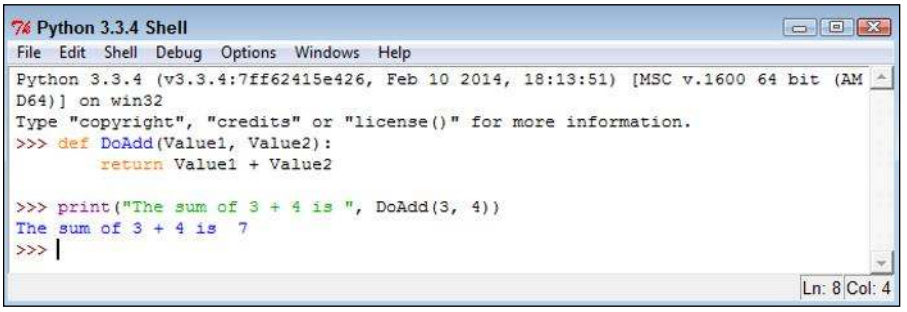

Figure 6-9: Return values can make your functions even more useful.

Comparing function output

You use functions with return values in a number of ways. For example, the previous section of this chapter shows how you can use functions to provide input for another function. You use functions to perform all sorts of tasks. One of the ways to use functions is for comparison purposes. You can actually create expressions from them that define a logical output.

To see how this might work, use the `DoAdd()` function from the previous section. Type **print("3 + 4 equals 2 + 5 is ", (DoAdd(3, 4) == DoAdd(2, 5)))** and press Enter. You see the truth value of the statement that 3 + 4 equals 2 + 5, as shown in Figure 6-10. The point is that functions need not provide just one use or that you view them in just one way. Functions can make your code quite versatile and flexible.

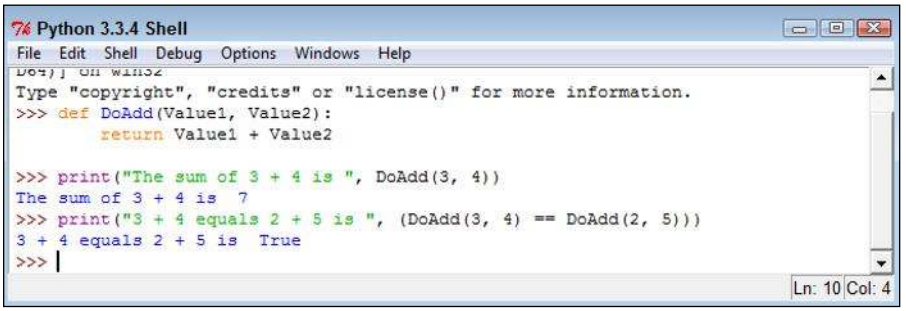

Figure 6-10: Use your functions to perform a wide variety of tasks.

Getting User Input

Very few applications exist in their own world — that is, apart from the user. In fact, most applications interact with users in a major way because computers are designed to serve user needs. To interact with a user, an application must provide some means of obtaining user input. Fortunately, the most commonly used technique for obtaining input is also relatively easy to implement. You simply use the `input()` function to do it.

The `input()` function always outputs a string. Even if a user types a number, the output from the `input()` function is a string. This means that if you are expecting a number, you need to convert it after receiving the input. The `input()` function also lets you provide a string prompt. This prompt is displayed to tell the user what to provide in the way of information.

The `Input01.py` file contains an example of using the `input()` function in a simple way. Here's the code for that example:

```
Name = input("Tell me your name: ")
print("Hello ", Name)
```

In this case, the `input()` function asks the user for a name. After the user types a name and presses Enter, the example outputs a customized greeting to the user. Try running this example at the command prompt or the Python Shell window. Figure 6-11 shows typical results when you input John as the username.

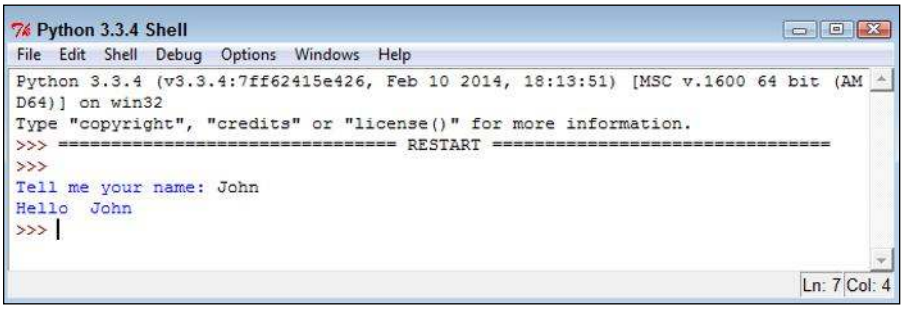

Figure 6-11: Provide a username and see a greeting as output.

You can use `input()` for other kinds of data; all you need is the correct conversion function. For example, the code in the `Input02.py` file provides one technique for performing such a conversion, as shown here:

```
ANumber = float(input("Type a number: "))
print("You typed: ", ANumber)
```

When you run this example, the application asks for a numeric input. The call to `float()` converts the input to a number. After the conversion, `print()` outputs the result. When you run the example using a value such as 5.5, you obtain the desired result.

It's important to understand that data conversion isn't without risk. If you attempt to type something other than a number, you get an error message, as shown in Figure 6-12. Chapter 9 helps you understand how to detect and fix errors before they cause a system crash.

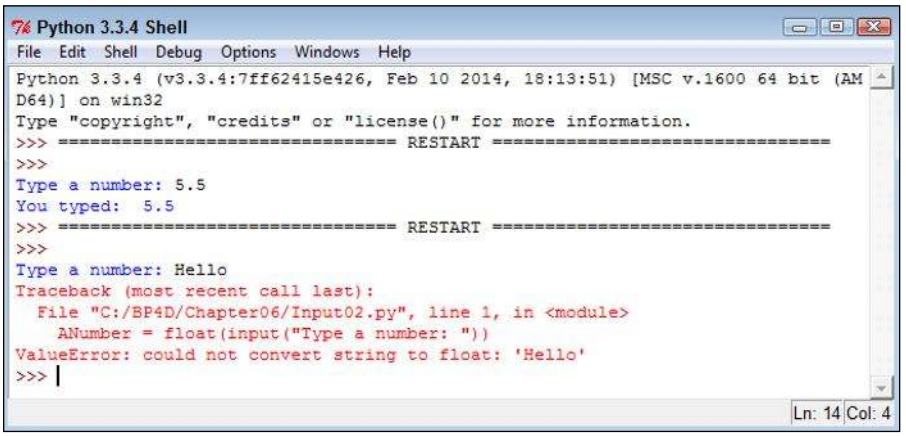

Figure 6-12: Data conversion changes the input type to whatever you need, but could cause errors.