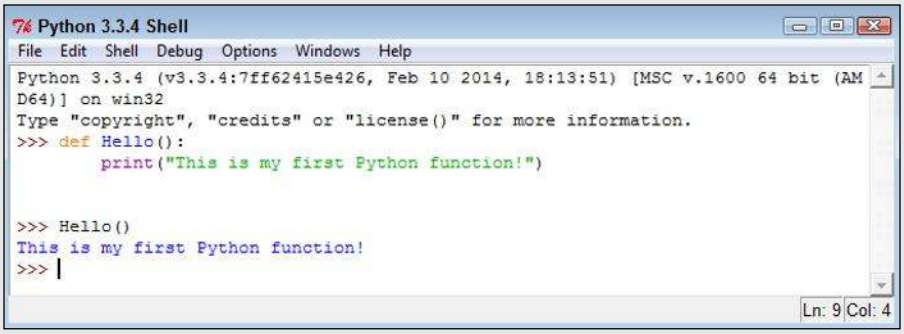
Part II

Talking the Talk



See an example of how you can combine functions and repetitive tasks at www.dummies.com/extras/beginningprogrammingwithpython.

In this part . . .

- See how to create variables to hold data.
- Create functions to make code easier to read.
- Tell your Python application to make a decision.
- Perform repeating tasks.
- Ensure that your application can deal with errors.

Chapter 5

Storing and Modifying Information

In This Chapter

- Understanding data storage
- Considering the kinds of data storage
- Adding dates and times to applications

Chapter 3 introduces you to CRUD, Create, Read, Update, and Delete — not that Chapter 3 contains cruddy material. This acronym provides an easy method to remember precisely what tasks all computer programs perform with information you want to manage. Of course, geeks use a special term for information — data, but either information or data works fine for this book.

In order to make information useful, you have to have some means of storing it permanently. Otherwise, every time you turned the computer off, all your information would be gone and the computer would provide limited value. In addition, Python must provide some rules for modifying information. The alternative is to have applications running amok, changing information in any and every conceivable manner. This chapter is about controlling information — defining how information is stored permanently and manipulated by applications you create.

Storing Information

An application requires fast access to information or else it will take a long time to complete tasks. As a result, applications store information in memory. However, memory is temporary. When you turn off the machine, the information must be stored in some permanent form, such as on your hard drive, a Universal Serial Bus (USB) flash drive, or a Secure Digital (SD) card. In addition, you must also consider the form of the information, such as whether it's a number or text. The following sections discuss the issue of storing information as part of an application in more detail.

Seeing variables as storage boxes

When working with applications, you store information in variables. A *variable* is a kind of storage box. Whenever you want to work with the information, you access it using the variable. If you have new information you want to store, you put it in a variable. Changing information means accessing the variable first and then storing the new value in the variable. Just as you store things in boxes in the real world, so you store things in variables (a kind of storage box) when working with applications.

Computers are actually pretty tidy. Each variable stores just one piece of information. Using this technique makes it easy to find the particular piece of information you need — unlike in your closet, where things from ancient Egypt could be hidden. Even though the examples you work with in previous chapters don't use variables, most applications rely heavily on variables to make working with information easier.

Using the right box to store the data

People tend to store things in the wrong sort of box. For example, you might find a pair of shoes in a garment bag and a supply of pens in a shoebox. However, Python likes to be neat. As a result, you find numbers stored in one sort of variable and text stored in an entirely different kind of variable. Yes, you use variables in both cases, but the variable is designed to store a particular kind of information. Using specialized variables makes it possible to work with the information inside in particular ways. You don't need to worry about the details just yet — just keep in mind that each kind of information is stored in a special kind of variable.

Python uses specialized variables to store information to make things easy for the programmer and to ensure that the information remains safe. However, computers don't actually know about information types. All that the computer knows about are 0s and 1s, which is the absence or presence of a voltage. At a higher level, computers do work with numbers, but that's the extent of what computers do. Numbers, letters, dates, times, and any other kind of information you can think about all come down to 0s and 1s in the computer system. For example, the letter *A* is actually stored as 01000001 or the number 65. The computer has no concept of the letter *A* or of a date such as 8/31/2014.

Defining the Essential Python Data Types

Every programming language defines variables that hold specific kinds of information, and Python is no exception. The specific kind of variable is called a *data type*. Knowing the data type of a variable is important because it tells you what kind of information you find inside. In addition, when you want to store information in a variable, you need a variable of the correct data type to do it. Python doesn't allow you to store text in a variable designed to hold numeric information. Doing so would damage the text and cause problems with the application. You can generally classify Python data types as numeric, string, and Boolean, although there really isn't any limit on just how you can view them. The following sections describe each of the standard Python data types within these classifications.

Putting information into variables

To place a value into any variable, you make an assignment using the assignment operator (=). Chapter 6 discusses the whole range of basic Python operators in more detail, but you need to know how to use this particular operator to some extent now. For example, to place the number 5 into a variable named `myVar`, you type **myVar = 5** and press Enter at the Python prompt. Even though Python doesn't provide any additional information to you, you can always type the variable name and press Enter to see the value it contains, as shown in Figure 5-1.

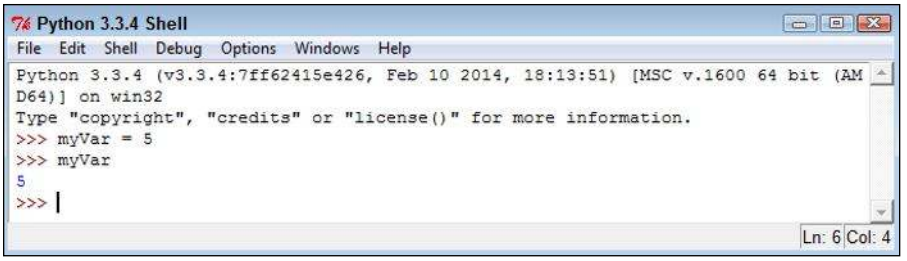


Figure 5-1: Use the assignment operator to place information into a variable.

Understanding the numeric types

Humans tend to think about numbers in general terms. We view 1 and 1.0 as being the same number — one of them simply has a decimal point. However, as far as we're concerned, the two numbers are equal and we could easily use them interchangeably. Python views them as being different kinds of numbers

because each form requires a different kind of processing. The following sections describe the integer, floating-point, and complex number classes of data types that Python supports.

Integers

Any whole number is an *integer.* For example, the value 1 is a whole number, so it's an integer. On the other hand, 1.0 isn't a whole number; it has a decimal part to it, so it's not an integer. Integers are represented by the `int` data type.

As with storage boxes, variables have capacity limits. Trying to stuff a value that's too large into a storage box results in an error. On most platforms, you can store numbers between –9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 within an `int` (which is the maximum value that fits in a 64-bit variable). Even though that's a really large number, it isn't infinite.

When working with the `int` type, you have access to a number of interesting features. Many of them appear later in the book, but one feature is the ability to use different numeric bases:

- **Base 2:** Uses only 0 and 1 as numbers.
- **Base 8:** Uses the numbers 0 through 7.
- **Base 10:** Uses the usual numeric system.
- **Base 16:** Is also called *hex* and uses the numbers 0 through 9 and the letters A through F to create 16 different possible values.

To tell Python when to use bases other than base 10, you add a 0 and a special letter to the number. For example, 0b100 is the value one-zero-zero in base 2. Here are the letters you normally use:

- **b:** Base 2
- **o:** Base 8
- **x:** Base 16

It's also possible to convert numeric values to other bases using the `bin()`, `oct()`, and `hex()` commands. So, putting everything together, you can see how to convert between bases using the commands shown in Figure 5-2. Try the command shown in the figure yourself so that you can see how the various bases work. Using a different base actually makes things easier in many situations, and you'll encounter some of those situations later in the book. For now, all you really need to know is that integers support different numeric bases.

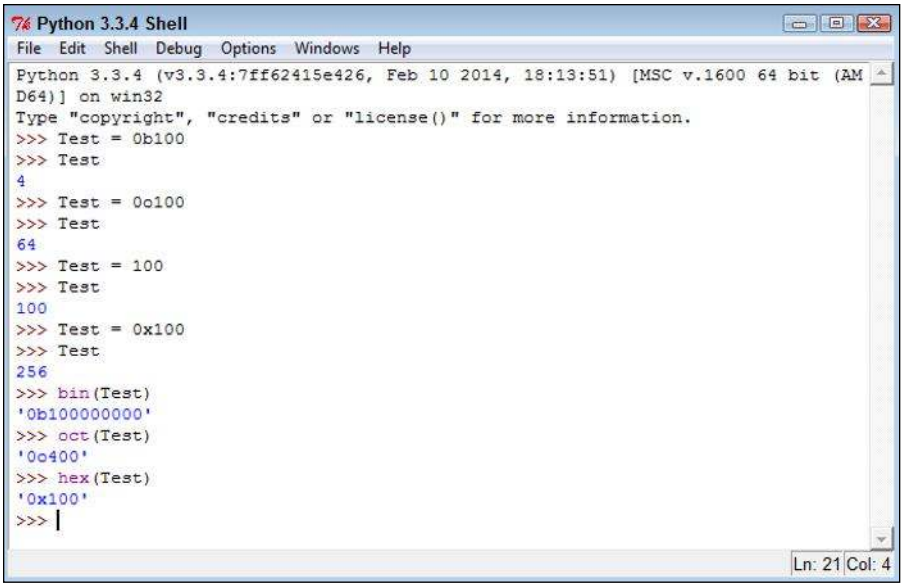


Figure 5-2: Integers have many interesting features, including the capability to use different numeric bases.

Floating-point values

Any number that includes a decimal portion is a floating-point value. For example, 1.0 has a decimal part, so it's a floating-point value. Many people get confused about whole numbers and floating-point numbers, but the difference is easy to remember. If you see a decimal in the number, then it's a floating-point value. Python stores floating-point values in the `float` data type.

Floating-point values have an advantage over integer values in that you can store immensely large or incredibly small values in them. As with integer variables, floating-point variables have a storage capacity. In their case, the maximum value that a variable can contain is $\pm 1.7976931348623157 \times 10^{308}$ and the minimum value that a variable can contain is $\pm 2.2250738585072014 \times 10^{-308}$ on most platforms.

When working with floating-point values, you can assign the information to the variable in a number of ways. The two most common methods are to provide the number directly and to use scientific notation. When using scientific notation, an *e* separates the number from its exponent. Figure 5-3 shows both methods of making an assignment. Notice that using a negative exponent results in a fractional value.

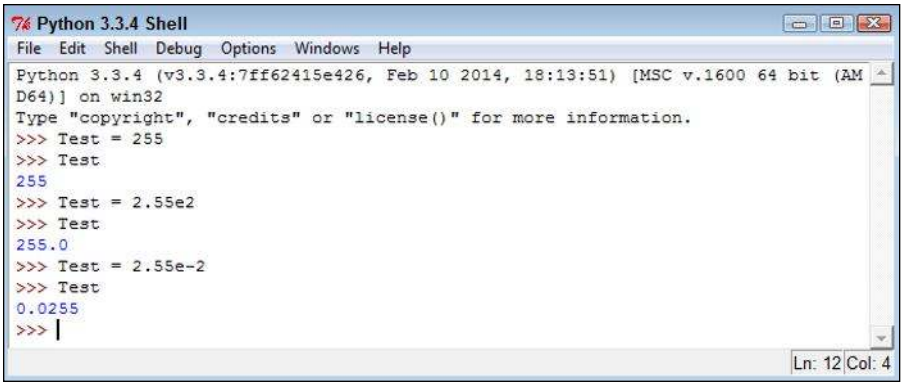


Figure 5-3: Floating-point values provide multiple assignment techniques.

Complex numbers

You may or may not remember complex numbers from school. A *complex number* consists of a real number and an imaginary number that are paired together. Just in case you've completely forgotten about complex numbers, you can read about them at `http://www.mathsisfun.com/numbers/complex-numbers.html`. Real-world uses for complex numbers include:

- Electrical engineering
- Fluid dynamics
- Quantum mechanics
- Computer graphics
- Dynamic systems

Complex numbers have other uses, too, but this list should give you some ideas. In general, if you aren't involved in any of these disciplines, you probably won't ever encounter complex numbers. However, Python is one of the few languages that provides a built-in data type to support them. As you progress through the book, you find other ways in which Python lends itself especially well to science and engineering.

The imaginary part of a complex number always appears with a *j* after it. So, if you want to create a complex number with 3 as the real part and 4 as the imaginary part, you make an assignment like this:

```
myComplex = 3 + 4j
```

If you want to see the real part of the variable, you simply type **myComplex.real** at the Python prompt and press Enter. Likewise, if you want to see the imaginary part of the variable, you type **myComplex.imag** at the Python prompt and press Enter.

Understanding the need for multiple number types

A lot of new developers (and even some older ones) have a hard time understanding why there is a need for more than one numeric type. After all, humans can use just one kind of number. To understand the need for multiple number types, you have to understand a little about how a computer works with numbers.

An integer is stored in the computer as simply a series of bits that the computer reads directly. A value of 0100 in binary equates to a value of 4 in decimal. On the other hand, numbers that have decimal points are stored in an entirely different manner. Think back to all those classes you slept through on exponents in school — they actually come in handy sometimes. A floating-point number is stored as a sign bit (plus or minus), *mantissa* (the fractional part of the number), and *exponent* (the power of 2). (Some texts use the term *significand* in place of mantissa — the terms are interchangeable.) To obtain the floating-point value, you use the equation:

```
Value = Mantissa * 2^Exponent
```

At one time, computers all used different floating-point representations, but they all use the IEEE-754 standard now. You can read about this standard at `http://grouper.ieee.org/groups/754/`. A full explanation of precisely how floating-point numbers work is outside the scope of this book, but you can read a fairly understandable description at `http://www.cprogramming.com/tutorial/floating_point/understanding_floating_point_representation.html`. Nothing helps you understand a concept like playing with the values. You can find a really interesting floating-point number converter at `http://www.h-schmidt.net/FloatConverter/IEEE754.html`, where you can click the individual bits (to turn them off or on) and see the floating-point number that results.

As you might imagine, floating-point numbers tend to consume more space in memory because of their complexity. In addition, they use an entirely different area of the processor — one that works more slowly than the part used for integer math. Finally, integers are precise, as contrasted to floating-point numbers, which can't precisely represent some numbers, so you get an approximation instead. However, floating-point variables can store much larger numbers. The bottom line is that decimals are unavoidable in the real world, so you need floating-point numbers, but using integers when you can reduces the amount of memory your application consumes and helps it work faster. There are many trade-offs in computer systems, and this one is unavoidable.

Understanding Boolean values

It may seem amazing, but computers always give you a straight answer! A computer will never provide "maybe" as output. Every answer you get is either `True` or `False`. In fact, there is an entire branch of mathematics called Boolean algebra that was originally defined by George Boole (a super-geek of his time) that computers rely upon to make decisions. Contrary to common belief, Boolean algebra has existed since 1854 — long before the time of computers.

Determining a variable's type

Sometimes you might want to know the variable type. Perhaps the type isn't obvious from the code or you've received the information from a source whose code isn't accessible. Whenever you want to see the type of a variable, use the `type()` method. For example, if you start by placing a value of 5 in `myInt` by typing **myInt = 5** and pressing Enter, you can find the type of `myInt` by typing **type(myInt)** and pressing Enter. The output will be `<class 'int'>`, which means that `myInt` contains an `int` value.

When using Boolean value in Python, you rely on the `bool` type. A variable of this type can contain only two values: `True` or `False`. You can assign a value by using the `True` or `False` keywords, or you can create an expression that defines a logical idea that equates to true or false. For example, you could say, `myBool = 1 > 2`, which would equate to `False` because 1 is most definitely not greater than 2. You see the `bool` type used extensively in the book, so don't worry about understanding this concept right now.

Understanding strings

Of all the data types, strings are the most easily understood by humans and not understood at all by computers. If you have read the previous chapters in this book, you have already seen strings used quite. For example, all the example code in Chapter 4 relies on strings. A *string* is simply any grouping of characters you place within double quotes. For example, `myString = "Python is a great language."` assigns a string of characters to `myString`.

The computer doesn't see letters at all. Every letter you use is represented by a number in memory. For example, the letter *A* is actually the number 65. To see this for yourself, type **ord("A")** at the Python prompt and press Enter. You see 65 as output. It's possible to convert any single letter to its numeric equivalent using the `ord()` command.

Because the computer doesn't really understand strings, but strings are so useful in writing applications, you sometimes need to convert a string to a number. You can use the `int()` and `float()` commands to perform this conversion. For example, if you type **myInt = int("123")** and press Enter at the Python prompt, you create an `int` named `myInt` that contains the value `123`. Figure 5-4 shows how you can perform this task and validate the content and type of `myInt`.

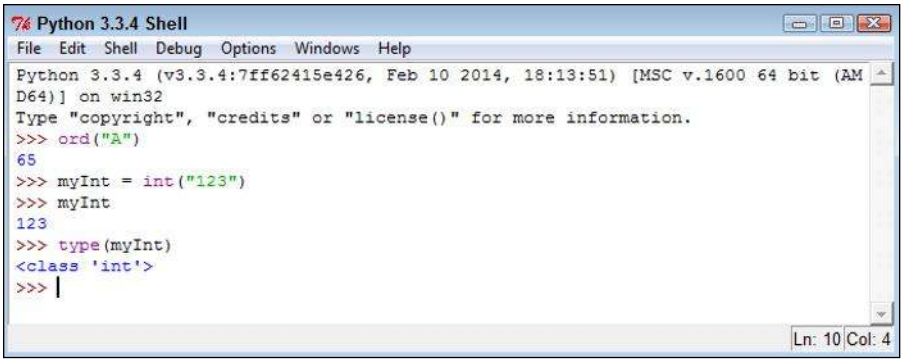


Figure 5-4: Converting a string to a number is easy using the `int()` and `float()` commands.

You can convert numbers to a string as well by using the `str()` command. For example, if you type **myStr = str(1234.56)** and press Enter, you create a string containing the value `"1234.56"` and assign it to `myStr`. Figure 5-5 shows this type of conversion and the test you can perform on it. The point is that you can go back and forth between strings and numbers with great ease. Later chapters demonstrate how these conversions make a lot of seemingly impossible tasks quite doable.

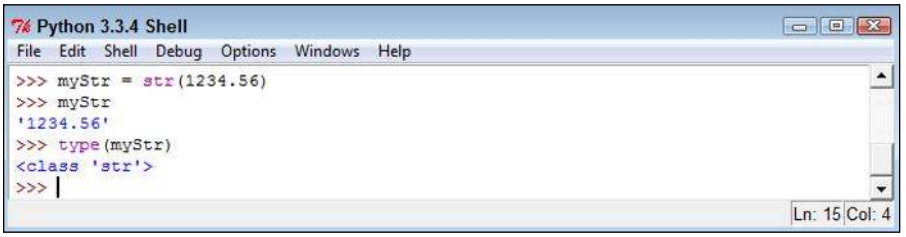


Figure 5-5: It's possible to convert numbers to strings as well.

Working with Dates and Times

Dates and times are items that most people work with quite a bit. Society bases almost everything on the date and time that a task needs to be or was completed. We make appointments and plan events for specific dates and times. Most of our day revolves around the clock. Because of the time-oriented nature of humans, it's a good idea to look at how Python deals with interacting with dates and time (especially storing these values for later use). As with everything else, computers understand only numbers — the date and time don't really exist.

To work with dates and times, you need to perform a special task in Python. When writing computer books, chicken-and-egg scenarios always arise, and this is one of them. To use dates and times, you must issue a special `import`

`datetime` command. Technically, this act is called *importing a module,* and you learn more about it in Chapter 10. Don't worry how the command works right now — just use it whenever you want to do something with date and time.

Computers do have clocks inside them, but the clocks are for the humans using the computer. Yes, some software also depends on the clock, but again, the emphasis is on human needs rather than anything the computer might require. To get the current time, you can simply type **datetime.datetime.now()** and press Enter. You see the full date and time information as found on your computer's clock (see Figure 5-6).

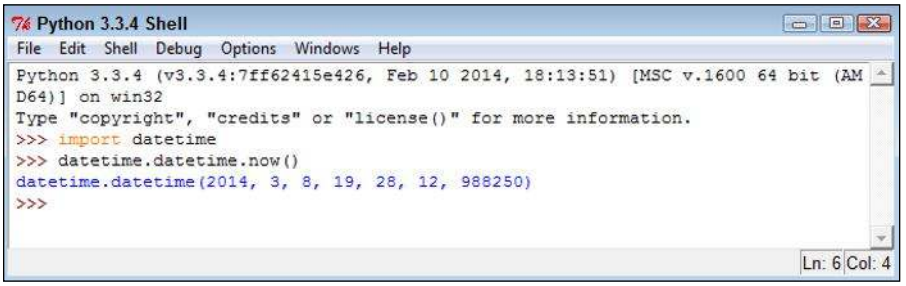


Figure 5-6: Get the current date and time using the `now()` command.

You may have noticed that the date and time are a little hard to read in the existing format. Say that you want to get just the current date, in a readable format. It's time to combine a few things you discovered in previous sections to accomplish that task. Type **str(datetime.datetime.now().date())** and press Enter. Figure 5-7 shows that you now have something a little more usable.

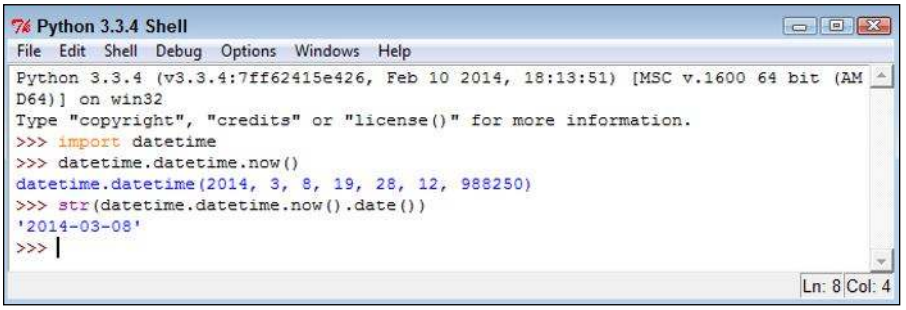


Figure 5-7: Make the date and time more readable using the `str()` command.

Interestingly enough, Python also has a `time()` command, which you can use to obtain the current time. You can obtain separate values for each of the components that make up date and time using the `day`, `month`, `year`, `hour`, `minute`, `second`, and `microsecond` values. Later chapters help you understand how to use these various date and time features to keep application users informed about the current date and time on their system.