1

Programming Basics and Strings

This chapter is a gentle introduction to the practice of programming in Python. Python is a very rich language with many features, so it is important to learn to walk before you learn to run. Chapters 1 through 3 provide a basic introduction to common programming ideas, explained in easily digestible paragraphs with simple examples.

If you are already an experienced programmer interested in Python, you may want to read this chapter quickly and take note of the examples, but until Chapter 3 you will be reading material with which you've probably already gained some familiarity in another language.

If you are a novice programmer, by the end of this chapter you will learn the following:

- ❑ Some guiding principles for programming
- ❑ Directions for your first interactions with a programming language — Python.

The exercises at the end of the chapter provide hands-on experience with the basic information that you have learned.

How Programming is Different from Using a Computer

The first thing you need to understand about computers when you're programming is that you control the computer. Sometimes the computer doesn't do what you expect, but even when it doesn't do what you want the first time, it should do the same thing the second and third time — until you take charge and change the program.

The trend in personal computers has been away from reliability and toward software being built on top of other, unreliable, software. The results that you live with might have you believing that computers are malicious and arbitrary beasts, existing to taunt you with unbearable amounts of extra work and various harassments while you're already trying to accomplish something. However, after you've learned how to program, you gain an understanding of how this situation has come to pass, and perhaps you'll find that you can do better than some of the programmers whose software you've used.

Note that programming in a language like Python, an *interpreted* language, means that you are not going to need to know a whole lot about computer hardware, memory, or long sequences of 0s and 1s. You are going to write in text form like you are used to reading and writing but in a different and simpler language. Python is the language, and like English or any other language(s) you speak, it makes sense to other people who already speak the language. Learning a programming language can be even easier, however, because programming languages aren't intended for discussions, debates, phone calls, plays, movies, or any kind of casual interaction. They're intended for giving instructions and ensuring that those instructions are followed. Computers have been fashioned into incredibly flexible tools that have found a use in almost every business and task that people have found themselves doing, but they are still built from fundamentally understandable and controllable pieces.

Programming is Consistency

In spite of the complexity involved in covering all of the disciplines into which computers have crept, the basic computer is still relatively simple in principle. The internal mechanisms that define how a computer works haven't changed a lot since the 1950s when transistors were first used in computers.

In all that time, this core simplicity has meant that computers can, and should, be held to a high standard of consistency. What this means to you, as the programmer, is that anytime you tell a computer to metaphorically jump, you must tell it how high and where to land, and it will perform that jump — over and over again for as long as you specify. The program should not arbitrarily stop working or change how it works without you facilitating the change.

Programming is Control

Programming a computer is very different from creating a program, as the word applies to people in real life. In real life, you ask people to do things, and sometimes you have to struggle mightily to ensure that your wishes are carried out — for example, if you plan a party for 30 people and assign two of them to bring the chips and dip and they bring the drinks instead, it is out of your control.

With computers that problem doesn't exist. The computer does exactly what you tell it to do. As you can imagine, this means that you must pay some attention to detail to ensure that the computer does just what you want it to do.

One of the goals of Python is to program in *blocks* that enable you to think about larger and larger projects by building each project as pieces that behave in well-understood ways. This is a key goal of a programming style known as *object-oriented programming*. The guiding principle of this style is that you can create reliable pieces that still work when you piece them together, that are understandable, and that are useful. This gives you, the programmer, control over how the parts of your programs run, while enabling you to extend your program as the problems you're solving evolve.

Programming Copes with Change

Programs are run on computers that handle real-world problems; and in the real world, plans and circumstances frequently change. Because of these shifting circumstances, programmers rarely get the opportunity to create perfectly crafted, useful, and flexible programs. Usually, you can achieve only two of these goals. The changes that you will have to deal with should give you some perspective and lead you to program cautiously. With sufficient caution, you can create programs that know when they're being asked to exceed their capabilities, and they can fail gracefully by notifying their users that they've stopped. In the best cases, you can create programs that explain what failed and why. Python offers especially useful features that enable you to describe what conditions may have occurred that prevented your program from working.

What All That Means Together

Taken together, these beginning principles mean that you're going to be introduced to programming as a way of telling a computer what tasks you want it to do, in an environment where you are in control. You will be aware that sometimes accidents can happen and that these mistakes can be accommodated through mechanisms that offer you some discretion regarding how these conditions will be handled, including recovering from problems and continuing to work.

The First Steps

The absolute first step you need to take before you can begin programming in Python is to download and install Python version 3.1. Navigate to `www.python.org/download` and choose the newest version of Python. You will be taken to a page with instructions on how to download the appropriate version for your computer. For instance, if you are running Windows, it may say Windows x86 MSI Installer (3.0).

Programs are written in a form called *source code*. Source code contains the instructions that the language follows, and when the source code is read and processed, the instructions that you've put in there become the actions that the computer takes.

Just as authors and editors have specialized tools for writing for magazines, books, or online publications, programmers also need specialized tools. As a starting Python programmer, the right tool for the job is the Python IDLE GUI (graphical user interface).

Once the download is finished, double-click it to run the program. Your best bet is to accept the default prompts Python offers you. This process may take a few minutes, depending on your system.

After setup is complete, you will want to test to make sure it is installed properly. Click the Windows Start menu and go to All Programs. You will see Python 3.0 in the menu. Choose IDLE (Python GUI) and wait for the program to load.

Once IDLE launches, type in "Test, test, testing" and press the Enter key. If Python is running correctly, it should return the value

```
'Test, test, testing'
```

in blue letters and with single quotes (I'll get more into this soon). Congratulations — you have successfully installed Python and are well on your way to becoming a programming guru.

Installing Python 3.1 on Non-Windows Systems

If you are the proud owner of a Mac and are running Mac OS X, you are in luck; it comes with Python installed. Unfortunately, it may not be the most up-to-date version. For security and compatibility purposes, I would suggest logging on to `www.python.org/download/mac`. Check to see that your Mac OS X version is the right version for the Python you are installing.

If you have a Linux computer, you may also already have Python installed, but again, it may be an earlier version. I would once more suggest you go to the Python website to find the latest version (and of course, the one appropriate to your system). The website `www.python.org/download` should have instructions on how to download the right version for your computer.

Using the Python Shell

Before starting to write programs, you'll need to learn how to experiment with the Python shell. For now, you can think of the Python shell as a way to peer within running Python code. It places you inside of a running instance of Python, into which you can feed programming code; at the same time, Python will do what you have asked it to do and will show you a little bit about how it responds to its environment. Because running programs often have a *context* — things that you as the programmer have tailored to your needs — it is an advantage to have the shell because it lets you experiment with the context you have created.

Now that you have installed Python version 3.1, you can begin to experiment with the shell's basic behavior. For starters, type in some text:

```
>>>"Hello World. You will never see this."
```

Note that typing the previous sentence into the shell didn't actually do anything; nothing was changed in the Python environment. Instead, the sentence was evaluated by Python, to determine what, if anything, you wanted Python to do. In this case, you merely wanted it to read the text.

Although Python didn't technically do anything with your words, it did give some indication that it read them. Python indicated this by displaying the text you entered (known as a *string*) in quotes. A *string* is a data type, and each data type is displayed differently by Python. As you progress through this book, you will see the different ways Python displays each one.

Beginning to Use Python — Strings

At this point, you should feel free to experiment with using the shell's basic behavior. Type some text, in quotes; for starters, you could type the following:

```
>>> "This text really won't do anything"
"This text really won't do anything"
```

You should notice one thing immediately: After you entered a quote ("), the Python shell changed the color of everything up to the quote that completed the sentence. Of course, the preceding text is absolutely true. It did nothing: It didn't change your Python environment; it was merely *evaluated* by the running Python instance, in case it did determine that in fact you'd told it to do something. In this case, you've asked it only to read the text you wrote, but doing this doesn't constitute a change to the environment.

However, you can see that Python indicated that it saw what you entered. It showed you the text you entered, and it displayed it in the manner it will always display a string — in quotes. As you learn about other *data types*, you'll find that Python has a way of displaying each one differently.

What is a String?

A *string* is one of several data types that exist within the Python language. A data type, as the name implies, is a category that a particular type of data fits into. Every type of data you enter into a computer is segregated into one of these data types, whether they be numbers or letters, as is the case in this scenario. Giving data a type allows the computer to determine how to handle the data. For instance, if you want the program to show the mathematical equation 1+1 on a screen, you have to tell it that it is text. Otherwise, the program will interpret the data as a mathematical equation and evaluate it accordingly.

You'll get more into the different data types and how important it is to define them in a later chapter. For now however, know that a string is a data type that consists of any character, be it a letter, number, symbol, or punctuation mark. Therefore, the following are all examples of strings:

"Hello, how are you?"

"1+1"

"I ate 4 bananas"

"!@#$%^&*()"

Why the Quotes?

When you type a string into Python, you do so by preceding it with quotes. Whether these quotes are single ('), double("), or triple(""") depends on what you are trying to accomplish. For the most part, you will use single quotes, because it requires less effort (you do not need to hold down the Shift key to create them). Note, however, that they are interchangeable with the double and even triple quotes.

Try typing in some strings. After you type in each sentence, press the Enter key to allow Python to evaluate your statement.

Try It Out Entering Strings with Different Quotes

Enter the following strings, keeping in mind the type of quotes (single or double) and the ends of lines (use the Enter key when you see that the end of a line has been reached):

```
>>> "This is a string using a double quote"
'This a string using a double quote'
>>> 'This is a string with a single quote'
'This is a string with a single quote'
>>> """This string has three quotes
look at what it can do!"""
'This string has three quotes\nlook at what it can do!'
>>>
```

In the preceding examples, although the sentences may look different to the human eye, the computer is interpreting them all the same way: that is, as a string. There is a true purpose to having three different quoting methods, which is described next.

Why Three Types of Quotes?

The reasoning behind having three types of quotes is fairly simple. Let's say that you want to use a contraction in your sentence, as I have just done. If you type a sentence such as "I can't believe it's not butter" into the shell, nothing much happens, but when you actually try to get the program to *use* that string in any way, you will get an error message. To show you what I mean, the following section introduces you to the `print()` function.

Using the print() Function

A function in Python (and every other programming language) is a tool developers use to save time and make their programs more efficient. Instead of writing the same code over and over again, they store it in a function, and then call upon that function when they need it. Don't worry too much about functions at the moment; they are covered in greater detail later on. For now, it is enough to know what the term means and how it relates to programming.

The `print()` function is used whenever you want to print text to the screen. Try the following example in your Python shell:

```
>>> print("Hello World!")
```

When you press Enter, you should see the following:

```
Hello World!
```

You will want to note several things here. First, as you were entering in the `print()` function, a pop-up as shown in Figure 1-1 appeared, showing you the various options available to you within the function:

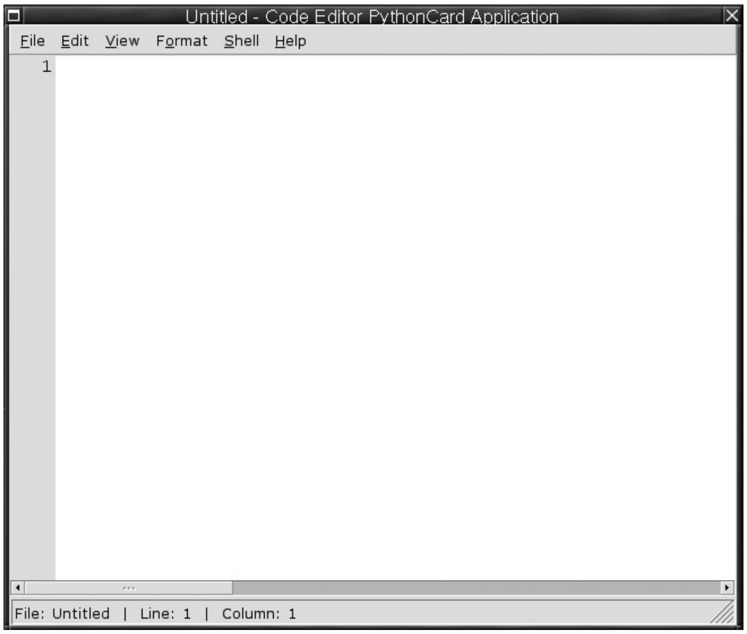

Untitled - Code Editor PythonCard Application
File Edit View Format Shell Help
File: Untitled | Line: 1 | Column: 1

Figure 1-1

Second, the text once more appeared in blue on the next line, but this time without quotation marks around it. This is because unlike in the previous examples, Python actually *did* something with the data.

Congratulations, you just wrote your first program!

Understanding Different Quotes

Now that you know how to use the `print()` function, you can begin to work with the different types of quotes discussed earlier in this chapter. Try the examples from earlier:

```
>>> print('This is a string using a single quote!')
This is a string using a single quote!
>>>print("This is a string using a double quote!")
This is a string using a double quote!
>>>print("""This string has three quotes!
Look at what it can do!""")
This string has three quotes
Look at what it can do!
```

In this example, you see that the single quote (') and double quote (") are interchangeable *in those instances.* However, when you want to work with a contraction, such as *don't*, or if you want to quote someone quoting something, observe what happens:

```
>>>print("I said, "Don't do it")
```

When you press Enter to execute the function, you will get the error message: `SyntaxError: invalid syntax (<pyshell#10>, line 1)`. I know what you are thinking — "What happened? I thought double and single quotes are interchangeable." Well, they are for the most part. However, when you try to mix them, it can often end up in a syntax error, meaning that your code has been entered incorrectly, and Python doesn't know what the heck you are trying to say.

What really happens here is that Python sees your first double quote and interprets that as the beginning of your string. When it encounters the double quote before the word *Don't*, it sees it as the end of the string. Therefore, the letters *on* make no sense to Python, because they are not part of the string. The string doesn't begin again until you get to the single quote before the *t*.

There is a simple solution to this, known as an escape. Retry the preceding code, adding an escape character to this string:

```
>>>print("I said, \"Don't do it")
I said, "Don't do it
```

This time, your code worked. When Python saw the backslash (`\`), or escape character, it knew to treat the double quote as a character, and not as a data type indicator. As you may have noticed, however, there is still one last problem with this line of code. See the missing double quote at the end of your results? To get Python to print the double quote at the end of the sentence, you simply add another escape character and a second double quote, like so:

```
>>>print("I said, \"Don't do it\"")
I said, "Don't do it"
```

Finally, let's take a moment to discuss the triple quote. You briefly saw its usage earlier. In that example, you saw that the triple quote allows you to write some text on multiple lines, without being processed until you close it with another triple quote. This technique is useful if you have a large amount of data that you do not wish to print on one line, or if you want to create line breaks within your code. Here, in the next example, you write a poem using this method:

```
>>>print("""Roses are red
Violets are blue
I just printed multiples lines
And you did too!""")
Roses are red
Violets are blue
I just printed multiple lines
And you did too!
```

There is another way to print text on multiple lines using the newline (`\n`) escape character, which is the most common of all the escape characters. I'll show it to you here briefly, and come back to discuss it in more depth in a later chapter. Try this code out:

```
>>>print("Roses are red \n Violets are blue \n
I just printed multiple
lines \n And you did too!")
Roses are red
Violets are blue
I just printed multiple lines
And you did too!
```

As you can see, the results are the same. Which you use is up to you, but the newline escape is probably more efficient and easier to read.

Putting Two Strings Together

There comes a time in every programmer's life when they have to combine two or more strings together. This is known as *concatenation*. For example, let's say that you have a database consisting of employees' first and last names. You may, at some point, wish to print these out as one whole record, instead of as two. In Python, each of these items can be treated as one, as shown here:

```
>>>"John"
'John'
>>>"Everyman"
'Everyman'
```

Try It Out Using + to Combine Strings

You can use several different methods to join distinct strings together. The first is by using the mathematical approach:

```
>>> "John" + "Everyman"
'JohnEveryman'
```

You could also just skip the + symbol altogether and do it this way:

```
>>>"John" "Everyman"
JohnEveryman
```

As you can see from these examples, both strings were combined; however, Python read the statement literally, and as such, there is no space between the two strings (remember: Python now views them as one string, not two!). So how do you fix this? You can fix it in two simple ways. The first involves adding a space after the first string, in this manner:

```
>>>"John " "Everyman"
John Everyman
```

I do not recommend this approach, however, because it can be difficult to ascertain that you added a space to the end of *John* if you ever need to read the code later in the future, say, when you are bleary-eyed and its four in the morning. The other approach is to simply use a separator, like so:

```
>>>"John" + " " + "Everyman"
John Everyman
```

Other reasons exist why you should use this method instead of simply typing in a space that have to do with database storage, but that is covered Chapter 14. Note that you can make any separator you like:

```
>>>"John" + "." + "Everyman"
John.Everyman
```

Joining Strings with the Print() Function

By default, the `print()` function is a considerate fellow that inserts the space for you when you print more than one string in a sentence. As you will see, there is no need to use a space separator. Instead, you just separate every string with a comma (,):

```
>>>Print("John" , "Everyman")
John Everyman
```

Putting Strings Together in Different Ways

Another way to specify strings is to use a *format specifier*. It works by putting in a special sequence of characters that Python will interpret as a placeholder for a value that will be provided by you. This may initially seem like it's too complex to be useful, but format specifiers also enable you to control what the displayed information looks like, as well as a number of other useful tricks.

Try It Out Using a Format Specifier to Populate a String

In the simplest case, you can do the same thing with your friend, John Q.:

```
>>> "John Q. %s" % ("Public")
'John Q. Public'
```

How It Works

The `%s` is known as a format specifier, specifically for strings. As the discussion on data types continues throughout this book, you take a look at several more, each specific to its given data type. Every specifier acts as a placeholder for that type in the string; and after the string, the `%` sign outside of the string indicates that after it, all of the values to be inserted into the format specifier will be presented there to be used in the string.

You may notice the parentheses. This tells the string that it should expect to see a sequence that contains the values to be used by the string to populate its format specifiers.

A simpler way to think of it is to imagine that the `%s` is a storage bin that holds the value in the parentheses. If you want to do more than one value, you would simply add another format specifier, in this manner:

```
>>>"John %s%s" % ("Every" , "Man")
John Everyman
```

These sequences are an integral part of programming in Python, and as such, they are covered in greater detail later in this book. For now, just know that every format specification in a string has to have an element that matches it in the sequence that is provided to it. Each item in the sequence are strings that must be separated by commas.

So why do they call it a format specifier if you store data in it? The reason is that it has multiple functions; being a container is only one of them. The following example shows you how to not only store data with the format specifier, but specify how that data will be displayed as well.

Try It Out More String Formatting

In this example, you tell the format specifier how many characters to expect. Try the following code and watch what happens:

```
>>> "%s %s %10s" % ("John" , "Every", "Man")
'John Every        Man'
>>> "%-5s %s %10s" % ("John" , "Every", "Man")
John     Every          Man
```

How It Works

In the first line of code, the word *Man* appears far away from the other words; this is because in your last format specifier, you added a 10, so it is expecting a string with ten characters. When it does not find ten (it only finds three . . . M-a-n) it pads space in between with seven spaces.

In the second line of code you entered, you will notice that the word *Every* is spaced differently. This occurs for the same reason as before — only this time, it occurred to the left, instead of the right. Whenever you right a negative (–) in your format specifier, the format occurs to the left of the word. If there is just a number with no negative, it occurs to the right.

Summary

In this chapter you learned how to install Python, and how to work with the Python GUI (IDLE), which is a program written in Python for the express purpose of editing Python programs. In addition to editing files, this "shell" allows you to experiment with simple programming statements in the Python language.

Among the things you learned to do within the shell are the basics of handling strings, including string concatenation, as well as how to format strings with format specifiers, and even storing strings within that same `%s` format specifier. In addition, you learned to work with multiple styles of quotes, including the single, double, and triple, and found out what the `\n` newline escape character was for.

Finally, you learned your very first function, `print()`, and wrote your first program, the Hello World standby, which is a time-honored tradition among programmers; it's similar to learning "Smoke on the Water" if you play guitar — it's the first thing you'll ever learn.

The key things to take away from this chapter are:

- ❑ Programming is consistency. All programs are created with a specific use in mind, and your user will expect the program not only to live up to that usage, but to work in exactly the same manner each and every time. If the user clicks a button and a print dialog box pops up, this button should always work in this manner.
- ❑ Programming is control. As a programmer, you control the actions your application can and cannot take. Even aspects of the program that seem random to the casual observer are, in fact, controlled by the parameters that you create.

- Programming copes with changes. Through repeated tests, you can ensure that your program responds appropriately to the user, even when they ask the program to do something you did not develop it to do.
- Strings are a data type, or simply put, a category of data. These strings allow you to interact with the user in a plethora of ways, such as printing text to the window, accepting text from the user, and so forth. A string can consist of any letter, number, or special character.
- The `print()` function allows you to print text to the user's screen. It follows the syntax: `print("Here is some text").`

Exercises

1. In the Python shell, type the string, `"Rock a by baby,\n\ton the tree top,\t\when the wind blows\n\t\t\t the cradle will drop."` Feel free to experiment with the number of `\n` and `\t` escape sequences to see how this affects what gets displayed on your screen. You can even try changing their placement. What do you think you are likely to see?
2. In the Python shell, use the same string indicated in Exercise 1, but this time, display it using the `print()` function. Once more, try differing the number of `\n` and `\t` escape sequences. How do you think it will differ?

2

Numbers and Operators

From our first breath of air, we are raised to use numbers. As a baby, we use them for estimating distance as we begin to crawl and, eventually, stand. As time progresses, we branch out and use them on a more conscious level, such as when we purchase a beverage or calculate our monthly budget. Whether you are one year old or 90, to some degree you are familiar with numbers. Indeed, numbers are such a familiar concept that you probably don't notice the many different ways in which you use them depending on their context.

In this chapter, you are re-introduced to numbers and some of the ways in which Python works with them, including basic arithmetic and special string format specifiers for its different types of numbers.

In this chapter you learn:

- ❑ To be familiar with the different basic categories of numbers that Python uses.
- ❑ To be familiar with the methods for using those numbers.
- ❑ The displaying and mixing the various number types.

Different Kinds of Numbers

If you have ever used a spreadsheet, you've noticed that the spreadsheet doesn't just look at numbers as *numbers* but as different kinds of numbers. Depending on how you've formatted a cell, the spreadsheet will have different ways of displaying the numbers. For instance, when you deal with money, your spreadsheet will show one dollar as `1.00`. However, if you're keeping track of the miles you've traveled in your car, you'd probably only record the miles you've traveled in tenths of a mile, such as 10.2. When you name a price you're willing to pay for a new house you probably only think to the nearest thousand dollars. At the large end of numbers, your electricity bills are sent to you with meter readings that come in at kilowatt hours, which are each one thousand watts per hour.

What this means in terms of Python is that, when you want to use numbers, you sometimes need to be aware that not all numbers relate to each other (as you see with imaginary numbers in this chapter), and sometimes you'll have to be careful about what kind of number you have and what you're trying to do with it. However, in general, you will use numbers in two ways: The first way will be to tell Python to repeat a certain action, and the second way will be to represent things that exist in the real world (that is, in your program, which is trying to model something in the real world). You will rarely have to think of numbers as anything besides simple numbers when you are counting things inside of Python. However, when you move on to trying to solve problems that exist in the real world — things that deal with money, science, cars, electricity, or anything else — you'll find yourself more aware about how you use numbers.

Numbers in Python

Python offers three different kinds of numbers with which you can work: *integers*, *floating-point numbers* (or *floats*), and *imaginary numbers*.

In previous versions of the language, Python had a different way of handling larger numbers. If a number ranged from –2,147,483,648 to +2,147,483,647, it was deemed an integer. Anything larger was promoted to a long. All that has changed, and the two types have now merged. Now, integers are described as a whole number, either positive or negative.

To determine the class of a number, you can use a special function that is built into Python, called `type`. When you use `type`, Python will tell you what kind of data you're looking at. Let's try this with a few examples.

Try It Out   Using Type with Different Numbers

In the Python shell, you can enter different numbers and see what `type` tells you about how Python sees them:

```
>>> type(1)
<class 'int'>
>>> type(2000)
<class 'int'>
>>> type(999999999999)
<class 'int'>
>>> type(1.0)
<class 'float'>
```

How It Works

Although in everyday life 1.0 is the same number as 1, Python will automatically perceive `1.0` as being a float; without the .0, the number 1 would be dealt with as the integer number one (which you probably learned as a whole number in grade school), which is a different kind of number.

In essence, the special distinction between a float and an integer is that a float has a component that is a fraction of 1. Numbers such as 1.01, 2.34, 0.02324, and any other number that contains a fractional component is treated as a floating-point number (except for imaginary numbers, which have rules of their own). This is the type that you would want to use for dealing with money or with things dealt with in partial quantities, like gasoline or pairs of socks. (There's always a stray single sock in the drawers, right?)

A Word to the Wise: Numbers can be Tricky

Experts in engineering, financial, and other fields who deal with very large and very small numbers (small with a lot of decimal places) need even more accuracy and consistency than what built-in types like floats offer. If you're going to explore these disciplines within programming, you should use the available *modules*, a concept introduced in Chapter 7, which are written to handle the types of issues pertinent to the field in which you're interested. At the very least, using modules that are written to handle high-precision floating-point values in a manner that is specifically different than the default behavior is worth investigating if you have the need for them.

The last type of number that Python offers is oriented toward engineers and mathematicians. It's the *imaginary number*, and you may remember it from school; it's defined as the square root of –1. Despite being named imaginary, it does have a lot of practical uses in modeling real-world engineering situations, as well as in other disciplines like physics and pure math. The imaginary number is built into Python so that it's easily usable by user communities who frequently need to solve their problems with computers. Having this built-in type enables Python to help them do that. If you happen to be one of those people, you will be happy to learn that you're not alone, and Python is there for you.

Try It Out Creating an Imaginary Number

The imaginary number behaves very much like a float, except that it cannot be mixed with a float. When you see an imaginary number, it will have the letter *j* trailing it:

```
>>> 12j
12j
```

How It Works

When you use the letter `j` next to a number and outside the context of a string (that is, not enclosed in quotes), Python knows that you've asked it to treat the number you've just entered as an imaginary number. When any letter appears outside of a string, it has to have a special meaning, such as this modifier, which specifies the type of number, or a named variables (which you see in Chapter 3), or another special name. Otherwise, the appearance of a letter by itself will cause an error!

You can combine imaginary and nonimaginary numbers to create complex numbers:

```
>>> 12j + 1
(1+12j)
>>> 12j + 1.01
(1.01+12j)
>>> type (12j + 1)
<class 'complex'>
```

You can see that when you try to mix imaginary numbers and other numbers, they are not added (or subtracted, multiplied, or divided); they're kept separate, in a way that creates a complex number. Complex numbers have a real part and an imaginary part, but an explanation of how they are used is beyond the scope of this chapter, although if you're someone who needs to use them, the complex number module (that word again!) is something that you can explore once you've gotten through Chapter 6. The module's name is `cmath`, for complex math. Complex numbers are discussed further in Chapter 19.

Program Files

By now you should be fairly comfortable using the Python shell and writing different lines of code within it. You've used it for all of the examples thus far, but now you are going to use it in a different manner. Instead of simply typing in single lines of code that disappear once you close the GUI, you are now going to create and save actual files that you can open and use again.

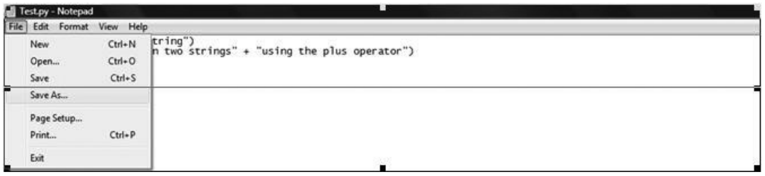
For the remainder of this chapter, you are encouraged to use the Python shell along with Notepad to create your very own files.

Try It Out By Typing the Following Text in Notepad

Enter the following into Notepad:

```
print("This is a basic string")
print("We learned to join two strings using " + "the plus operation")
```

Now that you have added some code to your editor, try and save it. First, go to File, then Save As (see Figure 2-1).



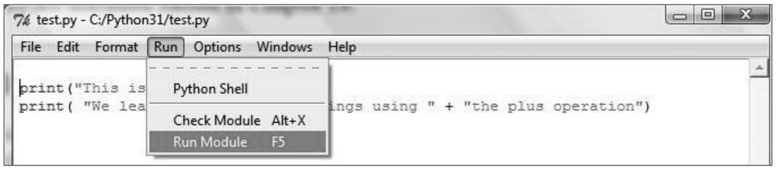
```
tring")
n two strings" + "using the plus operator")
```

Figure 2-1

A pop-up menu appears, prompting you for a name and directory in which to save your file. Python files use the extension `.py`, so always be sure to add it to the end of your file name, otherwise Notepad will save it as its default type, `.txt`. Give it the name `Test.py`. Next, navigate to the directory where Python is installed. Normally, this will be something along the lines of C:/Python31/. Click the Save button and you are all set.

After you've selected a file name and saved the file, you can reopen it. To run the `Test.py` *program, choose File ➪ Open from the Python shell, and choose the file you want to run (in this case, Test.py).*

The Python editor will now open. Click Run, choose Run Module (see Figure 2-2), and watch in amazement as your first program runs!



```
print("This is
print( "We lea                          ings using " + "the plus operation")
```

Figure 2-2

You will notice a few things. First, when you initially opened the `Test.py` file, Python took the liberty of highlighting your code in different colors. This makes functions and data types (and many other programming tidbits) easier to recognize. For instance, the `print()` function is in purple, whereas the string that comprises its value is green.

When you run this module, you no longer see the code, but its result instead, written out in blue text on your screen:

```
This is a basic string
We learned to join two strings using the plus operator
```

Do this a few more times with different strings, saving them in different files. Each one of these sessions is now available for you, and you can refer to them later.

Using the Different Types

Except for the basic integer, the other number types can grow to an unwieldy number of digits to look at and make sense of. Therefore, very often when these numbers are generated, you will see them in a format that is similar to scientific notation. Python will let you input numbers in this format as well, so it's a two-way street. There are many snags to using very large integers and floats. The topic is quite detailed and not necessarily pertinent to learning Python. If you want to know more about floating-point numbers in general, and what they really mean to a computer, the paper at `http://docs.sun.com/source/806-3568/ncg_goldberg.html` is a very good reference, although the explanation will only make sense to someone with prior experience with computers and numbers. Don't let that stop you from looking, though. It may be something you want to know about at some point in the future.

More commonly, you will be using integers and floats. It wouldn't be unusual to acquire a number from somewhere such as the date, the time, or information about someone's age or the time of day. After that data, in the form of a number, is acquired, you'll have to display it.

The usual method of doing this is to incorporate numbers into *strings*. You can use the format specifier method that was used in Chapter 1. It may make intuitive sense to you that you should also be able to use the + method for including a number in a string, but in fact this does not work, because deep down they are different types, and the + operator is intended for use only with two things of the same type: two strings, two numbers, or two other objects and types that you will encounter later. The definite exceptions are that floats and integers can be added together. Otherwise, you should expect that different types won't be combined with the + operation.

You are likely wondering why a string format specifier can be used to include a number, when a + can't. The reason is that the + operation relies on information contained in the actual items being added. Almost everything you use in Python can be thought of as an object with properties, and all of the properties combined define the object. One important property of every object is its type, and for now the important thing to understand about a type is that certain naturally understood things like the + operation work only when you perform them with two objects of compatible types. In most cases, besides numbers, compatible types should be thought of as the same type.

If you do want to use the + operation with numbers and strings (and doing this is usually a matter of style that you can decide for yourself), you can use a built-in function called `str` *that will transform, if possible, numbers into a string. It enables you to do things such as add strings and numbers into a single string. You can use* `str` *with most objects because most objects have a way of displaying themselves as strings. However, for the sake of consistency, you'll use string format specifiers for now.*

Try It Out Including Different Numbers in Strings

When you combined two strings in the first chapter by using a format specifier, you used the format specifier `%s`, which means "a string." Because numbers and strings have different types, you will use a different specifier that will enable your numbers to be included in a string:

```
>>> "Including an integer works with %%d like this: %d" % 10
'Including an integer works with %d like this: 10'
>>> "An integer converted to a float with %%f: %f" % 5
'An integer converted to a float with %f: 5.000000'
>>> "A normal float with %%f: %f" % 1.2345
'A normal float with %f: 1.234500'
>>> "A really large number with %%E: %E" % 6.789E10
'A really large number with %E: 6.789000E+10'
>>> "Controlling the number of decimal places shown: %.02f" % 25.101010101
'Controlling the number of decimal places shown: 25.10'
```

If you're wondering where you can use format specifiers, note that the last example looks very similar to the way we print monetary values, and, in fact, any program that deals with dollars and cents will need to have at least this much capability to deal with numbers and strings.

How It Works

Anytime you are providing a format specifier to a string, there may be options that you can use to control how that specifier displays the value associated with it. You've already seen this with the `%s` specifier in Chapter 1, where you could control how many characters were displayed. With numeric specifiers are also conventions regarding how the numbers of a particular type should be displayed. These conventions result in what you see when you use any of the numeric format specifiers.

Try It Out Escaping the % Sign in Strings

One other trick was shown before. If you want to print the literal string `%d` in your program, you achieve that in Python strings by using two `%` signs together. This is needed only when you also have valid format specifiers that you want Python to substitute for you in the same string:

```
>>> print("The %% behaves differently when combined with other letters, like
this: %%d %%s %%f %d" % 10)
The % behaves differently when combined with other letters, like this: %d %s
%f 10
```

How It Works

Note that Python pays attention to the combinations of letters and will behave correctly in a string that has both format specifiers as well as a double percent sign.

Basic Math

It's more common than not that you'll have to use the numbers in your program in basic arithmetic. Addition, subtraction, division, and multiplication are all built in. Addition and subtraction are performed by the + and – symbols.

Try It Out Doing Basic Math

You can enter basic arithmetic at the Python shell prompt and use it like a calculator. Like a calculator, Python will accept a set of operations, and when you press the Enter key, it will evaluate everything you've typed and give you your answer:

```
>>> 5 + 300
305
>>> 399 + 3020 + 1 + 3456
6876
>>> 300 - 59994 + 20
-59674
>>> 4023 - 22.46
4000.54
```

How It Works

Simple math looks about how you'd expect it to look. In addition to + and –, multiplication is performed by the asterisk, *, and division is performed by the forward slash, /. Multiplication and division may not be as straightforward as you'd expect in Python, because of the distinction between floating-point numbers and whole numbers.

In previous versions of Python, as numbers became larger, they would be promoted from int to long. However, in Python 3.1, these two types have merged and there is no longer a need for such promotion. Observe the following numbers and how Python promotes numbers once they become a certain size:

```
>>> 2000403030 * 392381727
784921595607432810
>>> 2000403030 * 3923817273929
7849215963933911604870
>>> 2e304 * 3923817273929
inf
>>> 2e34 * 3923817273929
7.8476345478579995e+46
```

Note that although Python can deal with some very large numbers, the results of some operations will exceed what Python can accommodate. The shorthand for infinity, `inf`, is what Python will return when a result is larger than what it can handle.

Before Python 3.1, division was a bit more interesting. Without help, Python would not coax one kind of number into another through division. Only when you had at least one number that was a floating-point component — that is, a period followed by a number — would floating-point answers be displayed. If two numbers that were normal integers or longs (in either case, lacking a component that specifies a value less than one, even if that is `.0`) were divided, the remainder would be discarded. This has since been fixed, and now Python will still display the decimals, unless told otherwise. Observe the following:

```
>>> 44 / 11
4.0
>>> 5.0/2.5
2.0
>>> 324/101
3.2079207920792081
>>> 324.5/102.9
3.1535471331389697
```

As you can see, if you divide an integer by another integer, it still shows as a floating point, even if there is no remainder. Likewise, dividing an integer by a floating point returns a floating-point number. Note, however, that even though the integer is displayed as a float in the preceding examples of 4.0 and 2.0, it is still, for all intents and purposes, an integer. However, the result of 324/101 is converted to a float.

Try It Out Using the Modulus Operation

There is one other basic operation of Python that you should be aware of: the remainder, or modulus operation. A new addition to Python is the ability to view the entire result of a piece of division (as you saw in the equation 324/101). Previously, if you wanted to know the remainder you had to use the modulus operator, because Python would show only the whole number portion of the answer. For 324/101, Python would have displayed 3. In some instances, believe it or not, you still need only the remainder portion of a division result. To find this part of the answer, you have to use the modulus operator, which is the `%`. Don't let this confuse you! The `%` means modulus only when it is used on numbers. When you are using strings, it retains its meaning as the format specifier. When something has different meanings in different contexts, it is called *overloading*, and it is very useful; but don't get caught by surprise when something behaves differently by design.

```
>>> 5 / 3
1.6666666666666667
>>> 5 % 3
2
```

How It Works

The preceding code indicates that 5 divided by 3 is 1.6666666666666667, and in the second example you learn that when you divide 5/3, you have a remainder of 2. One very useful task the modulus operator is used for is to discover whether one thing can be evenly divided by another, such as determining whether the items in one sequence will fit into another evenly (you learn more about sequences in Chapter 3). Here are some more examples that you can try out:

```
>>> 123 % 44
35
>>> 334 % 13
9
>>> 652 % 4
0
```

Some Surprises

You need to be careful when you are dealing with common floating-point values, such as money. Some things in Python are puzzling. For one thing, if you manipulate certain numbers with seemingly straightforward math, you may still receive answers that have extra values trailing them, such as the following:

```
>>> 4023 - 22.4
4000.5999999999999
```

The trailing nines could worry you, but they merely reflect the very high precision that Python offers. However, when you print or perform math, this special feature actually results in precise answers.

Try It Out — Printing the Results

Try actually printing the results, so that the preceding math with the unusual-looking results has its results displayed to a user, as it would from inside of a program:

```
>>> print("%f" % (4023 - 22.4))
4000.600000
```

How It Works

You may remember the earlier discussion regarding floating-point division, and how in Python 3.0, the entire equation is written out. Before, when you did the equation 5/3, you got the result 1.6666666666666667. But you might not want to display such a long string to the user. To truncate the answer, you can do so with the `%f` format specifier.

Try It Out %f Format Specifier

Try out the following code and observe the different ways Python handles floating-point mathematics and then how you can manipulate the results with formatting:

```
>>> print("%f" % (5/3))
1.666667
>>> print("%.2f" % (5/3))
1.67
>>> print("%f" % (415 * 20.2))
8383.000000
>>> print("%0.f" % (415 * 20.2))
8383
```

Floating-point numbers can be confusing. A complete discussion of floating-point numbers is beyond the scope of this book, but if you are experienced with computers and numbers and want to know more about floating-point numbers, read the paper at `http://docs.sun.com/source/806-3568/ncg_goldberg.html`. The explanation offered there should help round out this discussion.

Using Numbers

As you can see from the previous example, you can display numbers with the `print()` function by including the numbers into strings, for instance by using a format specifier. The important point is that you must determine how to display your numbers so that they mean what you intend them to mean, and that depends on knowing your application.

Order of Evaluation

When doing math, you may find yourself looking at an expression like 4*3+1/4–12. The puzzle you're confronted with is determining how you're going to *evaluate* this sort of expression and whether the way *you* would evaluate it is the same way that Python would evaluate it. The safest way to do this is to always enclose your mathematical expressions in parentheses, which will make it clear which math operations will be evaluated first.

Python evaluates these basic arithmetic operations as follows: Multiplication and division operations happen before addition and subtraction, but even this can become confusing.

Try It Out Using Math Operations

When you're thinking about a particular set of mathematical operations, it can seem straightforward when you write it down (or type it in). When you look at it later, however, it can become confusing. Try these examples, and imagine them without the parentheses:

```
>>> (24 * 8)
192
>>> (24 * (8 + 3))
```

```
264
>>> (24 * (8 + 3 + 7.0))
432.0
>>> (24 * (8 + 3 + 7.0 + 9))
648.0
>>> (24 * (8 + 3 + 7.0 + 9))/19
34.10526315789474
>>> (24 * (8 + 3 + 7 + 9))/19
34.10526315789474
>>> (24 * (8 + 3 + 7 + 9))%19
2
```

Notice in the examples here how the presence of any floating-point numbers changes the entire equation to using floating-point numbers, and how removing any floating-point numbers causes Python to evaluate everything as integers, unless the result is a float.

How It Works

The examples are grouped in something that resembles the normal order of evaluation, but the parentheses ensure that you can be certain which groups of arithmetic operations will be evaluated first. The innermost (the most contained) are evaluated first, and the outermost last. Within a set of parentheses, the normal order takes place.

Number Formats

When you prepare strings to contain a number, you have a lot of flexibility. The following Try It Out shows some examples.

For displaying money, use a format specifier indicating that you want to limit the number of decimal places to two.

Try It Out Using Number Formats

Try this, for example. Here, you print a number as though you were printing a dollar amount:

```
>>> print("$%.02f" % 30.0)
$30.00
```

You can use a similar format to express values less than a cent, such as when small items are listed for sale individually. When you have more digits than you will print, notice what Python does:

```
>>> print("$%.03f" % 30.00123)
$30.001
>>> print("$%.03f" % 30.00163)
$30.002
>>> print("%.03f" % 30.1777)
30.178
>>> print("%.03f" % 30.1113)
30.111
```

How It Works

As you can see, when you specify a format requiring more accuracy than you have asked Python to display, it will not just cut off the number. It will do the mathematically proper rounding for you as well.

Mistakes Will Happen

While you are entering these examples, you may make a mistake. Obviously, there is nothing that Python can do to help you if you enter a different number; you will get a different answer than the one in this book. However, for mistakes such as entering a letter as a format specifier that doesn't mean anything to Python or not providing enough numbers in a sequence you're providing to a string's format specifiers, Python tries to give you as much information as possible to indicate what's happened so that you can fix it.

Try It Out Making Mistakes

To understand what's going on when a mistake happens, here are some examples you can try. Their full meanings are covered later, starting in Chapter 4, but in the meantime, you should know this:

```
>>> print("%.03f" % (30.1113, 12))
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: not all arguments converted during string formatting
```

How It Works

In the preceding code, there are more elements in the sequence (three in all) than there are format specifiers in the string (just two), so Python helps you out with a message. What's less than helpful is that this mistake would cause a running program to stop running, so this is normally an error condition, or an *exception*. The term *arguments* here refers to the format specifiers but is generally used to mean parameters that are required in order for some object to work. When you call a function that expects a certain number of values to be specified, each one of those anticipated values is called an argument.

This is something that programmers take for granted; this specialized technical language may not make sense immediately, but it will begin to feel right when you get used to it. Through the first ten chapters of this book, arguments will be referred to as *parameters* to make them less puzzling, because no one is arguing, just setting the conditions that are being used at a particular point in time. When you are programming, though, the terms are interchangeable.

Here is another potential mistake:

```
>>> print("%.03f, %f %d" % (30.1113, 12))
Traceback (most recent call last):
  File "<input>", line 1, in ?
TypeError: not enough arguments for format string
```

Now that you know what Python means by an argument, it makes sense. You have a format specifier and you don't have a value in the accompanying sequence that matches it; thus, there aren't enough parameters.

If you try to perform addition with a string and a number, you will also get an error:

```
>>> "This is a string" + 4
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

This should make sense because you've already read about how you can and can't do this. However, here is definite proof: Python is telling you clearly that it can't do what it has been asked to do, so now it's up to you to resolve the situation. (Hint: You can use the `str` function.)

```
>>> "This is a string""" + str(4)
'This is a string4'
```

Some Unusual Cases

Python offers one other feature with its numbers that is worth knowing about so that you understand it when you encounter it. The normal counting system that we use is called *base 10*, or *radix 10*. It includes numbers from 0 to 9. Numbers above that just involve combining 0 through 9. However, computers commonly represent the binary numbers they actually deal with in *base 8*, called *octal*, and *base 16*, also called *hexadecimal*. These systems are often used to give programmers an easier way to understand bytes of data, which often come in one and two chunks of 8 bits.

In addition, neither octal nor hexadecimal can be displayed as negative numbers. Numbers described in this way are said to be *unsigned*, as opposed to being *signed*. The sign that is different is the + or – sign. Normally, numbers are assumed to be positive, but if a number is a *signed type*, it can be negative as well. If a number is unsigned, it has to be positive; and if you ask for the display of a negative number but in a signed format string, you'll get unusual answers.

Try It Out Formatting Numbers as Octal and Hexadecimal

```
>>> print('Octal uses the letter "o" lowercase. %d %o' % (10,10))
Octal uses the letter "o" lowercase. 10 12
```

It may seem like a mistake that the second number printed is 12 when you've provided the string with a 10. However, octal only has 8 numbers (0 to 7), so from 0 to 10 in octal is 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 11.

```
>>> print('Hex uses the letter "x" or "X". %d %x %X' % (10, 10, 10))
Hex uses the letter "x" or "X". 10 a A
```

Here is another case that needs explaining. Hexadecimal uses numbers from 0 to 15, but because you run out of numbers at 9, hex utilizes the letters a–f; and the letters are lowercase if you used the format specifier `%x` and are capitalized if you used `%X`. Therefore, the numbers 0 to 20 in decimal are as follows in hex: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, 12, 13.

Summary

This chapter introduced you to numbers in Python, although it doesn't cover everything available. You've seen and used three kinds of built-in numbers that Python knows about: integers, floats, and imaginary numbers. You have learned how to use string format specifiers to allow you to include numbers in your strings, and you've formatted those numbers in strings with different styles.

An important thing to remember is that the format, or how the number is displayed in a string, doesn't change the value of the number. Floats remain floats even when they are printed as integers, and vice versa.

You've performed the major built-in arithmetic operations: addition, subtraction, multiplication, division, and modulus. You have learned that if integers are mixed with a float, the result is a float, or if two integers are divided, they may also return a float, where appropriate. If arithmetic is done with an integer or a float combined with an imaginary number, the result will be a *complex number* that separates the real component and the imaginary component. You've also learned about the `type` function, which enables you to determine what type of number you actually have.

Lastly, you generally use numbers in base 10, or radix 10. Computers in general, and Python in particular, can easily translate numbers to base 8, or octal, and base 16, or hexadecimal.

The key things to take away from this chapter are:

- ❑ There are three types of numbers in Python. Those are: integers, which are whole numbers (both negative and positive; floating-point numbers, which are any number with a decimal value; and imaginary number, which is the square-root of 1, and is used in the world of engineering and physics.
- ❑ The + operator, when used on strings, concatenates two or more strings together. For instance, if you write print ("Hello," + " how are you?"), the result will be one sentence: "Hello, how are you?"
- ❑ To convert a number to a string, you can use the str function.
- ❑ Dividing two integers can sometimes result in a floating-point number (i.e.; 3/2). Dividing an integer by a floating-point decimal will always result in a floating-point number.
- ❑ The modulus operator (%) is used to return the remainder in a division. For instance, 5 % 2 will return 1.
- ❑ Using parentheses in your calculations helps to ensure the proper order of evaluation.

Exercises

Do the following first three exercises in Notepad and save the results in a file called `ch2_exercises.py`. You can run it from within Python by opening the file and choosing Run Module.

1. In the Python shell, multiply 5 and 10. Try this with other numbers as well.
2. Print every number from 6 through 14 in base 8.
3. Print every number from 9 through 19 in base 16.
4. Try to elicit other errors from the Python interpreter — for instance, by deliberately misspelling `print` as `pinrt`. Notice how as you work on a file in the Python shell, it will display `print` differently than it does `pinrt`.

3

Variables — Names for Values

In the previous two chapters, you learned how Python views strings, integers, floats, and imaginary numbers and how they can be created and displayed. This chapter presents more examples that demonstrate how these data types can be used.

In this chapter you learn:

- ❑ To use names to store the types you already know as well as other basic types to which you will be introduced.
- ❑ How to work with different types of objects that you haven't learned about yet. *Variables* and new, different types — specifically, you will become better acquainted with *lists*, *tuples*, and *dictionaries*.
- ❑ What a *reference* is and have some experience in using references.
- ❑ To get the most out of this chapter, you should type the examples yourself and alter them to see what happens.

Referring to Data — Using Names for Data

It's difficult to always write strings and numbers explicitly throughout a program because it forces you to remember everything. The exacting memory that computers have enable them to remember far more details than people can, and taking advantage of that capability is a huge part of programming. However, to make using data more flexible and easy, you want to give the data names that can be used to refer to them.

Try It Out Assigning Values to Names

These names are commonly called *variables*, which indicates that the data to which they refer can vary (it can be changed), while the name remains the same. You'll see them referred to as *names* as well, because that is what you are presented with by Python.

```
>>> first_string = "This is a string"
>>> second_string = "This is another string"
>>> first_number = 4
>>> second_number = 5
>>> print ("The first variables are %s, %s, %d, %d" % (first_string,
second_string, first_number, second_number))
The first variables are This is a string, This is another string, 4, 5
```

How It Works

You can see that you can associate a name with a value — either a string or an integer — by using the equals (=) sign. The name that you use doesn't relate to the data to which it points in any direct sense (that is, if you name it "number," that doesn't actually have to mean that it holds a number).

```
>>> first_string = 245
>>> second_number = "This isn't a number"
>>> print(first_string)
245
>>> print(second_number)
"This isn't a number"
```

Notice that you did not need to use quotations when you just wanted to print out the value inside of a variable. Had you put quotations around the variable inside the `print()` function, it would have printed out the name of the variable, instead of its contents, seeing it as a string and not an actual variable. The benefit of being able to name your data is that you can decide to give it a name that means something. It is always worthwhile to give your data a name that reminds you of what it contains or how you will use it in your program. If you were to inventory the lightbulbs in your home, you might want a piece of your program to contain a count of the lightbulbs in your closets and another piece to contain a count of those actually in use:

```
>>> lightbulbs_in_closet = 10
>>> lightbulbs_in_lamps = 12
```

As lightbulbs are used, they can be moved from the closet into the lamps, and a name can be given to the number of lightbulbs that have been thrown out this year, so that at the end of the year you have an idea of what 'you've bought, what you have, and what 'you've used; and when you want to know what you still have, you have only to refer to `lightbulbs_in_closet` or `lightbulbs_in_lamps`.

When you have names that relate to the value stored in them, 'you've created an informal index that enables you to look up and remember where you put the information that you want so that it can be easily used in your program.

Changing Data Through Names

If your data is a number or a string, you can change it by using the operations you already know you can do with them.

Try It Out Altering Named Values

Every operation you've learned for numbers and strings can be used with a variable name so that you can treat them exactly as if they were the numbers they referenced:

```
>>> proverb = "A penny saved"
>>> proverb = proverb + " is a penny earned"
>>> print(proverb)
A penny saved is a penny earned
>>> pennies_saved = 0
>>> pennies_saved = pennies_saved + 1
>>> print(pennies_saved)
1
print(pennies_saved + 1)
2
```

How It Works

Whenever you combine named values on the right-hand side of an equals sign, the names will be operated on as though you had presented Python with the values referenced by the names, even if the same name is on the left-hand side of the equals sign. When Python encounters a situation like that, it will first evaluate and find the result of the operations on the right side and then assign the result to the name on the left side. That way, there's no confusion about how the name can exist on both sides — Python will do the right thing.

Copying Data

The name that you give data is only a name. It's how you refer to the data that you're trying to access. This means that more than one name can refer to the same data:

```
>>> pennies_saved=1
>>> pennies_earned = pennies_saved
>>> print(pennies_earned)
1
```

When you use the = sign again, you are referring your name to a new value that you've created, and the old value will still be pointed to by the other name:

```
>>> pennies_saved = pennies_saved + 1
>>> print(pennies_saved)
2
>>> print(pennies_earned)
1
```

Names You Can't Use and Some Rules

Python uses a few names as special built-in words that it *reserves* for special use to prevent ambiguity. The following words are reserved by Python and can't be used as the names for data:

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

In addition, the names for data cannot begin with numbers or most non-alphabet characters (such as commas, plus or minus signs, slashes, and so on), with the exception of the underscore character. The underscore is allowed and even has a special meaning in some cases (specifically with classes and modules, which you see in Chapter 6 and later).

You will see a number of these special reserved words in the remaining discussion in this chapter. They're important when you are using Python to do various tasks.

Using More Built-in Types

Beside strings and numbers, Python provides four other important basic types: tuples, lists, sets, and dictionaries. These four types have a lot in common because they all allow you to group more than one item of data together under one name. Each one also gives you the capability to search through them because of that grouping. These groupings are indicated by the presence of enclosing parentheses (), square brackets [], and curly braces {}.

When you write a program, or read someone else's program, it is important to pay attention to the type of enclosing braces when you see groupings of elements. The differences among {}, [], and () are important.

Tuples — Unchanging Sequences of Data

In Chapters 1 and 2, you saw *tuples* (rhymes with supple) being used when you wanted to assign values to match more than one format specifier in a string. *Tuples* are a sequence of values, each one accessible individually, and a tuple is a basic type in Python. You can recognize tuples when they are created because they're surrounded by parentheses:

```
>>> print("A %s %s %s %s" % ("string", "filled", "by a", "tuple"))
A string filled by a tuple
```

Try It Out Creating and Using a Tuple

Tuples contain references to data such as strings and numbers. However, even though they refer to data, they can be given names just like any other kind of data:

```
>>> filler = ("string", "filled", "by a", "tuple")
>>> print("A %s %s %s %s" % ("string", "filled", "by a", "tuple"))
A string filled by a tuple
```

Note that you can also print out the values in the tuple by simply calling upon it in the `print()` function. Try the following code and observe the results:

```
>>> filler = ("string", "filled", "by a", "tuple")
>>> print(filler)
('string', 'filled', 'by a ', 'tuple')
```

As you can see, the four parts that made up the tuple were returned. This technique is useful if you ever want to see the individual parts that make up your tuple.

How It Works

You can see in the example that `filler` is treated exactly as though its data — the tuple with strings — were present and being used by the string to fill in its format specifiers because the tuple was treated exactly as though you had typed in a sequence to satisfy the format specification.

You can access a single value inside of a tuple. The value referred to by each *element* can be accessed directly by using the *dereference* feature of the language. With tuples, you dereference the value by placing square brackets after the name of the tuple, counting from zero to the element that you're accessing. Therefore, the first element is 0, the second element is 1, the third element is 2, and so on until you reach the last element in the tuple:

```
>>> a = ("first", "second", "third")
>>> print("The first element of the tuple is %s" % a[0])
The first element of the tuple is first
>>> print("The second element of the tuple is %s" % a[1])
The second element of the tuple is second
>>> print("The third element of the tuple is %s" % a[2])
The third element of the tuple is third
```

A tuple keeps track of how many elements it contains, and it can tell you when you ask it by using the built-in function `len`:

```
 >>> print("%d" % len(a))
3
```

This returns the number of elements in the tuple (in this case 3), so you need to remember that the `len` function starts counting at 1, but when you access your tuple, because tuples are counted starting from zero, you must stop accessing at one less than the number returned by `len`:

```
>>> print(a[len(a) - 1])
Third
```

You can also have one element of a tuple refer to an entirely different tuple. In other words, you can create layers of tuples:

```
>>> b = (a, "b's second element")
>>>print(b)
(('first', 'second', 'third'), "b's second element")
```

Now you can access the elements of the tuple a by adding another set of brackets after the first one, and the method for accessing the second element is no different from accessing the first — you just add another set of square brackets.

Try It Out Accessing a Tuple Through Another Tuple

Re-create the a and b tuples so that you can look at how this works. When you have these layers of sequences, they are sometimes referred to as *multidimensional* because there are two layers that can be visualized as going down and across, like a two-dimensional grid for graph paper or a spreadsheet. Adding another one can be thought of as being three-dimensional, like a stack of blocks. Beyond that, though, visualizing this can give you a headache, and it's better to look at it as layers of data.

```
>>> a = ("first", "second", "third")
>>> b = (a, "b's second element")
>>> print("%s" %b[1])
b's second element
>>> print("%s" % b[0][0])
first
>>> print("%s" % b[0][1])
second
>>> print("%s" % b[0][2])
third
```

How It Works

In each case, the code works exactly as though you had followed the reference in the first element of the tuple named b and then followed the references for each value in the second layer tuple (what originally came from the tuple a). It's as though you had done the following:

```
>>> a = ("first", "second", "third")
>>> b = (a, "b's second element")
>>> layer2 = b[0]
>>> print(layer2[0])
'first'
>>> print(layer2[1])
'second'
>>> print(layer2[2])
'third'
```

Note that tuples have one oddity when they are created. To create a tuple with one element, you absolutely have to follow that one element with a comma:

```
>>> single_element_tuple = ("the sole element",)
```

Doing otherwise will result in the creation of a string, and that could be confusing when you try to access it later.

A tuple can have any kind of data in it, but after you've created one it can't be changed. It is *immutable*, and in Python this is true for a few types (for instance, strings are immutable after they are created; and operations on them that look like they change them actually create new strings).

Tuples are immutable because they are supposed to be used for ordered groups of things that will not be changed while you're using them. Trying to change anything in them will cause Python to complain with an error, similar to the errors you were shown at the end of Chapter 2:

```
>>> a[1] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> print("%s" % a[1])
second
```

You can see that the error Python returns when you try to assign a value to an element in the tuple is a *TypeError,* which means that this type doesn't support the operation you asked it to do (that's what the equals sign does — it asks the tuple to perform an action). In this case, you were trying to get the second element in `a` to refer to an integer, the number 3, but that's not going to happen. Instead, `a` remains unchanged.

An unrelated error will happen if you try to refer to an element in a tuple that doesn't exist. If you try to refer to the fourth element in `a`, you will get an error (remember that because tuples start counting their elements at zero, the fourth element would be referenced using the number three):

```
>>> a[3]
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
  a[3]
IndexError: tuple index out of range
```

Note that this is an *IndexError* and that the explanation of the error is provided (although it doesn't tell you the index value that was out of range, you do know that you tried to access an element using an index value that doesn't exist in the tuple). To fix this in a program, you would have to find out what value you were trying to access and how many elements were in the tuple. Python makes finding these errors relatively simple compared to many other languages that will fail silently.

Lists — Changeable Sequences of Data

Lists, like tuples, are sequences that contain elements referenced starting at zero. Lists are created by using square brackets:

```
>>> breakfast = [ "coffee", "tea", "toast", "egg" ]
```

Try It Out Viewing the Elements of a List

The individual elements of a list can be accessed in the same way as tuples. Like tuples, the elements in a list are referenced starting at 0 and are accessed in the same order from 0 until the end:

```
>>> count = 0
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is coffee
>>> count = 1
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is tea
>>> count = 2
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is toast
>>> count = 3
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is egg
```

How It Works

When you are accessing more than one element of a list, one after the other, it is essential to use a name to hold the value of the numbered position where you are in the list. In simple examples like this, you should do it to get used to the practice, but in practice, you will always do this. Most often, this is done in a loop to view every element in a sequence (see Chapter 4 for more about loops).

Here, you're manually doing the work of increasing the value referred to by `count` to go through each element in the breakfast list to pull out the special for four days of the week. Because you're increasing the count, whatever number is referred to by `count` is the element number in the breakfast list that is accessed.

The primary difference in using a list versus using a tuple is that a list can be modified after it has been created. The list can be changed at any time:

```
>>> breakfast[count] = "sausages"
>>> print("Today's breakfast is %s" % breakfast[count])
Today's breakfast is sausages
```

You don't just have to change elements that already exist in the list, you can also add elements to the list as you need them. You can add elements at the end by using the `append` method that is built in to the list type. Using `append` enables you to append exactly one item to the end of a list:

```
>>> breakfast.append("waffles")
>>> count = 4
>>> print ("Today's breakfast is %s" % breakfast[count])
Today's breakfast is waffles
```

If you want to add more than one item to the end of a list — for instance, the contents of a tuple or of another list — you can use the *extend* method to append the contents of a list all at once. The list isn't included as one item in one slot; each element is copied from one list to the other:

```
>>> breakfast.extend(["juice", "decaf", "oatmeal"])
>>> print(breakfast)
['coffee', 'tea', 'toast', 'egg', 'waffle', 'juice', 'decaf', 'oatmeal']
```

As with tuples, you can't ask for an element beyond the end of a list, but the error message is slightly different from a tuple because the error will tell you that it's a list index that's out of range, instead of a tuple index that's out of range:

```
>>> count = 8
>>> print("Today's breakfast is %s" % breakfast[count])
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
  print("Today's breakfast is %s" % breakfast[count])
IndexError: list index out of range
```

The length of an array can also be determined by using the `len` function. Just like tuples, lengths start at one, whereas the first element of a list starts at zero. It's important to always remember this.

Dictionaries — Groupings of Data Indexed by Name

A *dictionary* is similar to lists and tuples. It's another type of container for a group of data. However, whereas tuples and lists are indexed by their numeric order, dictionaries are indexed by *names* that you choose. These names can be letters, numbers, strings, or symbols — whatever suits you.

Try It Out Making a Dictionary

Dictionaries are created using the curly braces. To start with, you can create the simplest dictionary, which is an empty dictionary, and populate it using names and values that you specify one per line:

```
>>> menus_specials = {}
>>> menus_specials["breakfast"] = "Canadian ham"
>>> menus_specials["lunch"] = "tuna surprise"
>>> menus_specials["dinner"] = "Cheeseburger Deluxe"
```

How It Works

When you first assign to `menus_specials`, you're creating an empty dictionary with the curly braces. Once the dictionary is defined and referenced by the name, you may start to use this style of specifying the name that you want to be the index as the value inside of the square brackets, and the values that will be referenced through that index are on the right side of the equals sign. Because they're indexed by names that you choose, you can use this form to assign indexes and values to the contents of any dictionary that's already been defined.

When you're using dictionaries, the indexes and values have special names. Index names in dictionaries are called *keys*, and the values are called, well, *values*. To create a fully specified (or you can think of it as a completely formed) dictionary — one with *keys* and *values* assigned at the outset — you have to specify each key and its corresponding value, separated by a colon, between the curly braces. For example, a different day's specials could be defined all at once:

```
>>> menu_specials = {"breakfast" : "sausage and eggs",
... "lunch" : "split pea soup and garlic bread",
... "dinner": "2 hot dogs and onion rings"}
```

To print out all of the keys and values in a dictionary, simply place the name of the dictionary in the parameters of the `print()` function, as shown in the following code. To access any of the values, you use square brackets with the name of the key enclosed in the brackets. If the key is a string, the key has to be enclosed in quotes. If the key is a number (you can use numbers, too, making a dictionary look a lot like a list or a tuple), you need only the bare number.

```
>>>print(menu_specials)
{'lunch': 'split pea soup and garlic bread', 'breakfast': 'sausage and eggs',
'dinner': '2 hot dogs and onion rings'}
>>> print("%s" % menu_specials["breakfast"])
sausage and eggs
>>> print("%s" % menu_specials["lunch"])
split pea soup and garlic bread
>>> print("%s" % menu_specials["dinner"])
2 hot dogs and onion rings
```

If a key that is a string is accidentally not enclosed in quotes when you try to use it within square brackets, Python will try to treat it as a name that should be dereferenced to find the key. In most cases, this will raise an exception — a NameError — unless it happens to find a name that is the same as the string, in which case you will probably get an IndexError from the dictionary instead!

Try It Out Getting the Keys from a Dictionary

Dictionaries can tell you what all of their keys are, or what all of their values are, if you know how to ask them. The `keys` method will ask the dictionary to return all of its keys to you as a view so that you can examine them for the key (or keys) you are looking for, and the `values` method will return all of the values as a view.

```
>>> hungry=menu_specials.keys()
>>>print(list(hungry))
lunch
breakfast
dinner
>>>starving=menu_specials.value()
>>>print(list(starving))
split pea soup and garlic bread
sausage and eggs
2 hot dogs and onion rings
```

How It Works

Both the keys and values methods return views, which you can assign and use like any normal view. When you have the items in a view from the keys method, you can use the items in the view, which are keys, to get their matching values from that dictionary. Note that while a particular key will lead you to a value, you cannot start with a value and reliably find the key associated with it. You try to find the key when you know only a value; you need to exhaustively test all the possible keys to find a matching value, and even then, two different keys can have the same values associated with them.

In addition, the way that dictionaries work is that each key is different (you can't have two keys that are exactly the same), but you can have multiple duplicate values:

```
>>>menu={"breakfast" : "spam", "lunch" : "spam", "dinner": "Spam with a side
of Spam"}
>>>print(menu)
{'lunch': 'spam', 'breakfast': 'spam', 'dinner': 'Spam with a side of Spam'}
>>> menu.get("lunch")
'spam'
>>> menu.get("breakfast")
'spam'
```

As you can see, Python has no problem allowing you to see multiple values in different keys. However, watch what happens when you try the following code, whose purpose is to try and create *keys* with the *same* name:

```
>>> menu2 = {"breakfast" : "spam", "breakfast" : "ham", "dinner": "Spam with
a side of Spam:"}
>>>menu2.get("breakfast")
'ham'
```

What happened here? Although you did not get an error, there is still a mistake in your code. When you typed in the second key named `"breakfast"`, Python replaced the value in the first key with the same name, and replaced the value of the second key with the same name.

Treating a String Like a List

Python offers an interesting feature of strings. Sometimes, it is useful to be able to treat a string as though it were a list of individual characters. It's not uncommon to have extraneous characters at the end of a string. People may not recognize these, but computers will get hung up on them. It's also common to only need to look at the first character of a string to know what you want to do with it. For instance, if you had a list of last names and first names, you could view the first letter of each by using the same syntax that you would for a list. This method of looking at strings is called *slicing* and is one of the fun things about Python:

```
>>> last_names = [ "Douglass", "Jefferson", "Williams", "Frank", "Thomas" ]
>>> print("%s" % last_names[0])
Douglass
>>> print("%s" % last_names[0][0])
D
>>> print("%s" % last_names[1])
Jefferson
>>> print("%s" % last_names[1][0])
J
>>> print("%s" % last_names[2])
Williams
>>> print("%s" % last_names[2][0])
W
```

(continued)

(continued)

```
>>> print("%s" % last_names[3])
Frank
>>> print("%s" % last_names[3][0])
F
>>> print("%s" % last_names[4])
Thomas
>>> print("%s" % last_names[4][0])
T
```

For example, you can use the letter positioning of strings to arrange them into groups in a dictionary based on the first letter of the last name. You don't need to do anything complicated; you can just check to see which letter the string containing the name starts with and file it under that:

```
>>> by_letter = {}
>>> by_letter[last_names[0][0]] = last_names[0]
>>> by_letter[last_names[1][0]] = last_names[1]
>>> by_letter[last_names[2][0]] = last_names[2]
>>> by_letter[last_names[3][0]] = last_names[3]
>>> by_letter[last_names[4][0]] = last_names[4]
```

The `by_letter` dictionary will, thanks to string slicing, only contain the first letter from each of the last names. Therefore, `by_letter` is a dictionary indexed by the first letter of each last name. You could also make each key in `by_letter` reference a list instead and use the `append` method of that list to create a list of names beginning with that letter (if, of course, you wanted to have a dictionary that indexed a larger group of names, where each one did not begin with a different letter).

Remember that, like tuples, strings are *immutable*. When you are slicing strings, you are actually creating new strings that are copies of sections of the original string.

String Slicing is Very Useful

If you're new to programming, string slicing may seem like an unusual feature at first. Programmers who have used a lower-level language like C or C++ would have learned how to program viewing strings as special lists (and in Python you can also slice lists, as you'll see later), so for them this is natural. For you, it will be a very convenient tool once you've learned how to control repetition over lists in Chapter 4.

Special Types

Python has a handful of special types. You've seen them all, but they bear mentioning on their own: `None`, `True`, and `False` are all special built-in values that are useful at different times.

`None` is special because there is only one `None`. It's a name that no matter how many times you use it, it doesn't match any other object, just itself. When you use functions that don't have anything to return to you — that is, when the function doesn't have anything to respond with — it will return `None`.

`True` and `False` are special representations of the numbers 1 and 0. This prevents a lot of the confusion that is common in other programming languages where the truth value of a statement is arbitrary. For instance, in a UNIX shell (shell is both how you interact with the system, as well as a programming language), 0 is true and anything else is false. With C and Perl, 0 is false and anything else is true. However, in all of these cases, there are no built-in names to distinguish these values. Python makes this easier by explicitly naming the values. The names *True* and *False* can be used in elementary comparisons, which you'll see a lot; and in Chapter 4, you learn how these comparisons can dramatically affect your programs — in fact, they enable you to make decisions within your program.

```
>>> True
True
>>> False
False
>>> True == 1
True
>>> True == 0
False
>>> False == 1
False
>>> False == 0
True
>>> False > 0
False
>>>False < 1
True
```

Other Common Sequence Properties

The two types of sequences are tuples and lists; and as you've seen, in some cases strings can be accessed as though they were sequences as well. Strings make sense because you can view the letters in a string as a sequence.

Even though dictionaries represent a group of data, they are not sequences, because they do not have a specific ordering from beginning to end, which is a feature of sequences.

Referencing the Last Elements

All of the sequence types provide you with some shortcuts to make their use more convenient. You often need to know the contents of the final element of a sequence, and you can get that information in two ways. One way is to get the number of elements in the list and then use that number to directly access the value there:

```
>>> last_names = [ "Douglass", "Jefferson", "Williams", "Frank", "Thomas" ]
>>> len(last_names)
5
>>> last_element = len(last_names) - 1
>>> print("%s" % last_names[last_element])
Thomas
```

However, that method takes two steps; and as a programmer, typing it repeatedly in a program can be time-consuming. Fortunately, Python provides a shortcut that enables you to access the last element of a sequence by using the number –1, and the next-to-last element with –2, letting you reverse the order of the list by using negative numbers from –1 to the number that is the negative length of the list (–5 in the case of the `last_names` list).

```
>>> print("%s" % last_names[-1])
Thomas
>>> print("%s" % last_names[-2])
Frank
>>> print("%s" % last_names[-3])
Williams
```

Ranges of Sequences

You can take sections of a sequence and extract a piece from it, making a copy that you can use separately. The term for creating these groupings is called *slicing* (the same term used for this practice when you did it with strings). Whenever a slice is created from a *list* or a *tuple,* the resulting slice is the same type as the type from which it was created, and you've already seen this with strings. For example, a slice that you make from a *list* is a *list*, a slice you make from a *tuple* is a *tuple*, and the slice from a *string* is a *string.*

Try It Out Slicing Sequences

You've already sliced strings, so try using the same idea to slice tuples, lists, and strings and see what the results are side-by-side:

```
>>> slice_me = ("The", "next", "time", "we", "meet", "drinks", "are", "on", "me")
>>> sliced_tuple = slice_me[5:9]
>>> print(sliced_tuple)
('drinks', 'are', 'on', 'me')
>>> slice_this_list = ["The", "next", "time", "we", "meet", "drinks",
"are", "on", "me"]
>>> sliced_list = slice_this_list[5:9]
>>> print(sliced_list)
['drinks', 'are', 'on', 'me']
>>> slice_this_string = "The next time we meet, drinks are on me"
>>> sliced_string = slice_this_string[5:9]
>>> print(sliced_string)
'ext '
```

How It Works

In each case, using the colon to specify a slice of the sequence instructs Python to create a new sequence that contains just those elements.

Growing Lists by Appending Sequences

Suppose you have two lists that you want to join together. You haven't been shown a purposely built way to do that yet. You can't use `append` to take one sequence and add it to another. Instead, you will find that you have layered a sequence into your list:

```
>>> living_room = ("rug", "table", "chair", "TV", "dustbin", "shelf")
>>> apartment = []
>>> apartment.append(living_room)
>>> apartment
[('rug', 'table', 'chair', 'TV', 'dustbin', 'shelf')]
```

This is probably not what you want if you were intending to create a list from the contents of the tuple `living_room` that could be used to create a list of all the items in the `apartment`.

To copy all of the elements of a sequence, instead of using `append`, you can use the `extend` method of lists and tuples, which takes each element of the sequence you give it and inserts those elements into the list from which it is called:

```
>>> apartment = []
>>> apartment.extend(living_room)
>>> apartment
['rug', 'table', 'chair', 'TV', 'dustbin', 'shelf']
```

Using Lists to Temporarily Store Data

You'll often want to acquire data from another source, such as a user entering data or another computer whose information you need. To do that, it is best to put this data in a list so that it can be processed later in the same order in which it arrived.

However, after you've processed the data, you no longer need it to be in the list, because you won't need it again. Temporal (time-specific) information such as stock tickers, weather reports, or news headlines would be in this category.

To keep your lists from becoming unwieldy, a method called `pop` enables you to remove a specific reference to data from the list after you're done with it. When you've removed the reference, the position it occupied will be filled with whatever the next element was, and the list will be reduced by as many elements as you've popped.

Try It Out Popping Elements from a List

You need to tell `pop` which element it is acting on. If you tell it to work on element 0, it will pop the first item in its list, passing `pop` a parameter of 1 will tell it to use the item at position 1 (the second element in the list), and so on. The element `pop` acts on is the same number that you'd use to access the list's elements using square brackets (remember that the first value in a list is 0):

```
>>> todays_temperatures = [23, 32, 33, 31]
>>> todays_temperatures.append(29)
>>> todays_temperatures
[23, 32, 33, 31, 29]
```

```
>>> morning = todays_temperatures.pop(0)
>>> print("This mornings temperature was %.02f" % morning)
This mornings temperature was 23.00
>>> late_morning = todays_temperatures.pop(0)
>>> print("Todays late morning temperature was %.02f" % late_morning)
Todays late morning temperature was 32.00
>>> noon = todays_temperatures.pop(0)
>>> print("Todays noon temperature was %.02f" % noon)
Todays noon temperature was 33.00
>>> todays_temperatures
[31, 29]
```

How It Works

When a value is popped, if the action is on the right-hand side of an equals sign, you can assign the element that was removed to a value on the left-hand side, or just use that value in cases where it would be appropriate. If you don't assign the popped value or otherwise use it, it will be discarded from the list.

You can also avoid the use of an intermediate name, by just using `pop` to populate, say, a string format, because pop will return the specified element in the list, which can be used just as though you'd specified a number or a name that referenced a number:

```
>>> print("Afternoon temperature was %.02f" % todays_temperatures.pop(0))
Afternoon temperature was 31.00
>>> todays_temperatures
[29]
```

If you don't tell `pop` to use a specific element (0 in the examples) from the list it's invoked from, it will remove the last element of the list, not the first as shown here.

Working with Sets

Sets are similar to dictionaries in Python, except that they consist of only keys with no associated values. Essentially, they are a collection of data with no duplicates. They are very useful when it comes to removing duplicate data from data collections.

Sets come in two types: *mutable* and *immutable frozensets.* The difference between the two is that with a mutable set, you can add, remove, or change its elements, while the elements of an immutable frozenset cannot be changed after they have been initially set.

Try It Out Removing Duplicates

Here, you assign some values and remove the duplicates by assigning them to a set:

```
>>> alphabet = ['a','b', 'b', 'c', 'a', 'd', 'e']
>>> print(alphabet)
['a', 'b', 'b', 'c', 'a', 'd', 'e']
>>> alph2 = set(alphabet)
{'a', 'c', 'b', 'e', 'd'}
```

How It Works

The example works by taking the data collection, `alphabet`, and converting it to a set. Because sets do not allow duplicate values, the extra `'b'` and `'a'` characters are removed. It was then assigned to `alph2`, and printed to show the results.

Summary

In this chapter, you learned how to manipulate many core types that Python offers. These types are *tuples*, *lists*, *dictionaries*, *sets*, and three special types: None, True, and False. You've also learned a special way that strings can be treated like a sequence. The other sequence types are tuples and lists.

A *tuple* is a sequence of data that's indexed in a fixed numeric order, starting at zero. The references in the tuple can't be changed after the tuple is created, nor can it have elements added or deleted. However, if a tuple contains a data type that has changeable elements, such as a list, the elements of that data type are not prevented from changing. Tuples are useful when the data in the sequence is better off not changing, such as when you want to explicitly prevent data from being accidentally changed.

A *list* is another type of sequence, which is similar to a tuple except that its elements can be modified. The length of the list can be modified to accommodate elements being added using the `append` method, and the length can be reduced by using the `pop` method. If you have a sequence whose data you want to append to a list, you can append it all at once with the `extend` method of a list.

Dictionaries are yet another kind of indexed grouping of data. However, whereas lists and tuples are indexed by numbers, dictionaries are indexed by values that you choose. To explore the indexes, which are called *keys*, you can invoke the `keys` method. To explore the data that is referred to, called the *values*, you can use the `values` method. Both of these methods return lists.

Sets are a collection of items (0 or more), that contain no duplicates. In theory, they are similar to dictionaries, except that they only have keys, and no values associated with those keys. One use for sets is to remove any duplicates from a collection of data. They are also good at mimicking finite mathematical sets.

Other data types are `True`, `False`, and `None`. `True` and `False` are a special way of looking at 1 and 0, but when you want to test whether something is true or false, explicitly using the names `True` and `False` is always the right thing to do. `None` is a special value that is built into Python that only equals itself, and it is what you receive from functions that otherwise would not return any value (such as `True`, `False`, a string, or other values).

The key things to take away from this chapter are:

- ❑ Variables are names for data that let you refer to the data.
- ❑ You create a variable by using the syntax: variablename = "Some value".
- ❑ You can copy the value in one variable by assigning it to another: variablename = copyofvariablename.

- Tuples store more than piece of data and are unchangeable.
- Lists are also sequences of data, yet unlike tuples, you can change their value.
- A dictionary is similar to lists and tuples. It's another type of container for a group of data. However, whereas tuples and lists are indexed by their numeric order, dictionaries are indexed by names that you choose. These names can be letters, numbers, strings, or symbols — whatever suits you.

Exercises

Perform all of the following in the Python shell:

1. Create a list called `dairy_section` with four elements from the dairy section of a supermarket.
2. Print a string with the first and last elements of the `dairy_section` list.
3. Create a tuple called `milk_expiration` with three elements: the month, day, and year of the expiration date on the nearest carton of milk.
4. Print the values in the `milk_expiration` tuple in a string that reads "This milk carton will expire on 12/10/2009."
5. Create an empty dictionary called `milk_carton`. Add the following key/value pairs. You can make up the values or use a real milk carton:
    - `expiration_date`: Set it to the `milk_expiration` tuple.
    - `fl_oz`: Set it to the size of the milk carton on which you are basing this.
    - `Cost`: Set this to the cost of the carton of milk.
    - `brand_name`: Set this to the name of the brand of milk you're using.
6. Print out the values of all of the elements of the `milk_carton` using the values in the dictionary, and not, for instance, using the data in the `milk_expiration` tuple.
7. Show how to calculate the cost of six cartons of milk based on the cost of `milk_carton`.
8. Create a list called cheeses. List all of the cheeses you can think of. Append this list to the `dairy_section` list, and look at the contents of `dairy_section`. Then remove the list of cheeses from the array.
9. How do you count the number of cheeses in the cheese list?
10. Print out the first five letters of the name of your first cheese.