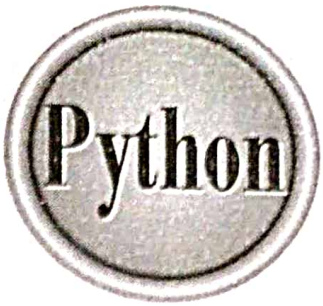
The Complete Reference



Chapter 3

Components of a Python Program

Now that you know how to write simple Python programs and how to execute them on the different implementations, it's time to take a closer look at the Python language. This chapter begins with a description of the basic components that make up a Python program. Since you have probably already done some form of programming, it's safe to describe programming as manipulating variables, since this is what all programs do regardless of their host language.

It's worth remembering throughout this chapter, and indeed during any Python programming exercise, that Python deals with *objects*. Although objects in Python may appear to be similar to the variables used in other languages (and we'll be making those comparisons in this chapter), all Python variables are, in truth, objects.

The Python language follows the same basic principles as many others:

- An individual application is made up of a number of files, which Python calls *modules*.
- Each module is composed of a number of statements.
- The statements create, use, and modify variables; Python creates object variables.

This chapter describes the two main areas make up a Python program; *Built-in Object Types* and the *Statement*. Since the logical flow is from the lowest common denominator, the object, to the highest, a module, we'll look at objects first and then look at statements. Chapter 6 looks at modules in depth, after you've mastered the basic components of the Python language.

Built-in Object Types

All languages support a number of built-in variable types. These are the building blocks used for all the other variable types available within the language. By combining or extending the basic variables, you can create some quite complex variables. For example, in C there is a basic type called **char**, which is a single character. By defining a **char** array, you create a text string, and by defining a two-dimensional array, you create an array or a string of strings.

Python supports a number of built-in object types. Unlike C, which approaches the problem of variables by supporting the lowest common denominator, Python supports a number of high-level variables that are more practical for most program solutions. Unlike other languages, Python variables are actually objects. By using objects at the basic storage level, the overall language is object-based. This has a number of advantages. You can use the same methods and functions to operate on different types of objects. Furthermore, because new objects inherit object methods, you can continue to use the same methods in the more complex objects that you create. You'll see some examples of this later in this chapter.

Just like other scripting languages, Python handles the creation, memory allocation, and the access routines that enable you to use the objects. This is the same model as offered by Visual Basic (VB) and Perl, but it's different from C, where you are responsible for allocating memory for the variables and information you need to store.

Python actually supports a wide variety of built-in object types that help to make your programs easier to write. Many of the basic types offered by Python would actually require a separate library or development process to be implemented before the object could be used. By providing these built-in, optimized versions, you can start programming right away with Python without worrying about the complexities of the objects you are using.

The use of built-in object types also has another advantage. The code for using built-in object types is highly optimized (and debugged) and has been developed over a number of years to be as efficient as possible. Although it is possible to write your own routines for implementing the same objects within your own applications, it is difficult to improve upon the versions supplied as built-in objects with the Python interpreter.

Python Objects and Other Languages

Python supports five main built-in object types and an external data type that is accessible just like the built-in objects. The built-in objects are Numbers, Strings, Lists, Dictionaries, and Tuples.

The primary external data type supported by Python is the file. Although it may seem odd to include files as a data type, Python supports access to files in much the same way that it supports access to the built-in object types. You'll see how this works and how the file and object types compare later in this chapter.

Some of the built-in object types will be familiar to you, such as numbers and strings and possibly lists, although Python supports a few different options on some of these objects. The other object types are probably unfamiliar to you with their current names. Table 3-1 offers a comparison of Python's built-in object types with data types offered by other languages.

Python	C/C++	Perl	Visual Basic
Number	int, long, float, double	scalar	double, integer, long, single, currency
String	char[] or char *	scalar	string
List	int[] or char[]	array	Any array type
Dictionary	N/A	hash or associative array	N/A
Tuple	*See Text*	*See Text*	*See Text*
File	FILE * or fd	FILE	No built-in type

Table 3-1. *Python and Other Language Data Types*

The *tuple* is a special type of list that cannot be modified. Since it's similar to a list, there is no reason why you can't use a normal array in C/C++, Perl, or VB. See the section "Tuples" later in this chapter for details on how tuples differ from normal lists.

Unlike other languages, it is the way in which the information is assigned to the object that defines the object type used to store the information. In C, you must explicitly define the type of the variable you are creating. In some languages, including Perl, the leading character in the variable name tells the language's interpreter what the object type is. With Python, the interpreter examines the information being placed into the object, and then creates or modifies the object to match the new type.

To see this in action, the following code shows different ways in which you can create objects. The operation is called an *assignment*, since you assign a value to a variable:

```
integer = 12345
float = 123.45
string = 'Hello'
list = [1, 'Two', 3]
dictionary = {'one':1, 'two':2, 'three':3}
tuple = (1, 'Two', 3)
```

In each case, Python identifies the information you are assigning to the variable and uses the correct object type for the data that needs to be stored. You'll learn more about the semantics of the assignment process when we describe each individual object type.

Operator Basics

It's impossible to talk about objects without also talking about the operators that can be used to operate on the objects. Because Python uses objects rather than variables to store information, you can actually use the same operators on a variety of different objects; Python automatically calls the corresponding method required to perform the operation on each object in question (see "Operator Overloading" later in this chapter).

Like other languages, Python follows the same basic mathematical notation and format when using operators on individual objects. Table 3-2 lists the operators and their precedence—operators at the bottom of the table have the highest precedence.

Many of these operators will be familiar to C/C++, Perl, and VB programmers, among others. For example, the expression **a+b** does exactly what you expect—it adds the value of variable **a** to variable **b**. Other operators, such as **lambda** and **in**, are specific to the Python language, and will be discussed when you learn about the individual semantics of each object type.

Parentheses

Despite the intonation in Table 3-2, the use of the parentheses, **()**, does not automatically imply the creation of a tuple. You can still use parentheses to modify the precedence

Operators	Description
x or y	Logical or (y is evaluated only if x is false)
lambda args: expression	Anonymous function
x and y	Logical and (y is evaluated only if x is true)
not x	Logical negation
<, <=, >, >=, ==, <>, !=	Comparison tests
is, is not	Identity tests
in, not in	Membership tests
x | y	Bitwise or
x^y	Bitwise exclusive or
x&y	Bitwise and
x<<y, x>>y	Shift x left or right by y bits
x+y, x-y	Addition/concatenation, subtraction
x*y, x/y, x%y	Multiplication/repetition, division, remainder/format
-x, +x, ~x	Unary negation, identity, bitwise complement
x[i], x[i:j], x.y, x(...)	Indexing, slicing, qualification, function call
(...), [...], {...}, `...`	Tuple, list, dictionary, conversion to string

Table 3-2. *Python Operators and Expression Precedence*

of an expression. As with other languages, any expression contained within parentheses is evaluated first—before the precedence shown in Table 3-2 goes into effect.

For example, the expression

```
result = 3*4+5
```

yields the value 17—correct according the precedence in Table 3-2. But note that the expression

```
result = 3*(4+5)
```

returns a value of 27.

Mixed Types in Expressions

Where applicable, Python follows the same basic rule as C/C++. If an expression contains mixed variable types, then the return value is of the most complex type contained within the expression. For example, the calculation

```
result = 3*2.5
```

sets the value of **result** to 7.5, a floating-point number, even though you started off with an integer and a floating-point value. Although this applies primarily to numerical calculations, there are some special cases with other object types. We'll discuss them separately.

Operator Overloading

The term *operator overloading* refers to the ability of a language to perform different operations on different object types when using the same operator. From a programming perspective, the ability of a language to support operator overloading allows for a simplified programming process. The + operator, for example, adds two numbers together when given two number objects. However, if given two strings, the strings are "added" together; in other words; the two strings are concatenated into a single string. Operator overloading also applies to other operators; the [] operators, for example, extract slices from strings, lists, and tuples.

It's important to understand the principle being used here—the operator performs the same *operation* on each object, even though the objects store and use their information in different ways.

Numbers

Python uses a simple object to hold a number. There are no restrictions on what type of number can be stored within a number object, unlike C, where you need to use a different data type to store integer and floating-point numbers. In addition to these basic types, Python also supports complex numbers and arbitrarily large integers.

As you already know, Python creates an object using the object type as determined from information that is assigned to the object when it is created. With number objects, the format of the number determines the method in which the information is stored. Therefore, you need to know how to introduce numerical constants into your Python programs.

Integer Constants

You can create integer number objects by supplying a sequence of numbers, as in the following example:

```
number = 1234
number = -1234
```

The objects created are integers, and they are actually stored internally as a C **long** data type, which is at least 32 bits long, and may be longer depending on the C compiler and processor being used. Also note that 0 is considered to be a number:

```
zero = 0
```

Again, this is as you would expect. It also follows that Python uses integer values when determining the logical value of an expression. As with other languages, 0 equates to a value of **false**, and any other number equates to **true**. This allows you to use integers for simple Boolean values without the need for an additional data type.

The use of only integer constants (or object values) in your expressions causes Python to use integers, instead of floating-point math routines. For highly repetitive calculations, the use of integers can increase performance. However, as described earlier, introducing a floating-point value into the expression causes the returned value to also be a floating-point number. See the "Floating-Point Constants" section later in this chapter for some examples.

Hexadecimal and Octal Constants

You can specify hexadecimal (base 16) and octal (base 8) constants using the same notation available in Perl and C/C++. That is, **0x** or **0X** prefixed to a number forces Python to interpret it as a hexadecimal number, while a single leading **0** (zero) indicates an octal number. For example, to set the value of an integer to decimal 255, you can use any one of the following statements:

```
decimal = 255
hexadecimal = 0xff
octal = 0377
```

Since these are simply integers, Python stores them in the same way as decimal integers. If you actually want to print an integer in octal or hexadecimal format, there is an interpolation method shown for formatting string objects that is described later in this chapter.

Long Integers

The built-in basic integer type is limited to a storage width of at least 32 bits. This means that the maximum number that you can represent is 2^{31}-1, since you must use one bit to allow for negative numbers. Although for many situations this is more than enough, there are times when you need to work with long integers. Python supports arbitrarily long integers—you can literally create an integer one thousand digits long and use it within Python as you would any other number.

To create a long integer, you must append an **l** or **L** to the end of the number constant, as in the following example:

```
long = 123456789123456789123456789123456789123456789123456789L
```

Once you have created a long integer, you can execute expressions as if they were normal numbers—the Python interpreter handles the complexities of dealing with the super-sized number. For example, the statements

```
long = 123456789123456789123456789123456789123456789123456789L
print long+1
```

produce the following output:

```
123456789123456789123456789123456789123456789123456790L
```

Or you can continue to do long integer math:

```
long = 123456789123456789123456789123456789123456789123456789L
print long+87654321987654321987654321987654321987654321987654321L
```

which generates

```
2111111111111111111111111111111111111111111111111111110L
```

Although Python uses these long integers as if they were normal integer values, the interpreter has to do a significant amount of extra work to support this option, even though support for such large numbers is written in C. If you can, use the built-in integer or floating-point types in preference to using long integer math.

Floating-Point Constants

Python supports the normal decimal point and scientific notation formats for representing floating-point numbers. For example, the following constants are all valid:

```
number = 1234.5678
number = 12.34E10
number = -12.34E-56
```

Internally, Python stores floating-point values as C doubles, giving the objects as much precision as possible. Note that there is no "long" floating-point variable.

Remember that floating-point or mixed floating-point and integer expressions return floating-point values. The easiest way to demonstrate this is to show the output from the same simple calculation—one using integer constants and the other floating-point constants:

```
>>> print 5/12
0
>>> print 5.0/12
0.416666666667
```

You can see from the preceding example that the first expression returns 0—the rounded-down version of 5/12. The second expression prints the expected decimal fraction.

For those situations where you are not using constants and therefore don't have overall control of the object types you are using for your calculations, you can force an integer value to be returned by using the built-in **int** function. This forces Python to imply integer status to the expression supplied to the function. See the section "Type Conversion" at the end of this section.

Complex Number Constants

Python employs the normal notation for supporting complex numbers—the real and imaginary parts are separated by a plus sign, and the imaginary number uses a single j or J suffix. For example, the following are examples of complex number constants:

```
cplx = 1+2j
cplx = 1.2+3.4j
```

Python uses two floating-point numbers to store the complex number, irrespective of the precision of the original. Because complex numbers are a separate entity within Python, the interpreter automatically performs complex math on expressions that include complex numbers.

Numeric Operators

Most of the operators in Table 3-2 apply to numbers. Table 3-3 contains a more explicit list of numeric operators used for calculations—these are all the familiar mathematical operations.

Operation	Description
x+y	Add x to y
x-y	Subtract y from x
x*y	Multiply x by y
x/y	Divide x by y
x**y	Raise x to the power of y
x%y	Modulo (returns the remainder of x/y)
-x	Unary minus
+x	Unary plus

Table 3-3. *Numeric Operators for all Number Types*

There are also a series of shift and bitwise operators that can be used for binary and bit math; these are listed in Table 3-4. Note that these operators can only be applied to integers; trying the operations on a floating-point number raises an exception.

In addition to these base operators, a series of augmented assignment operators was introduced with Python 2.0. For example, you can rewrite the fragment

```
a = a + 5
```

using an augmented assignment operator, as

```
a += 5
```

More clearly, the expression

```
x = x + y
```

can be rewritten as

```
x += y
```

The following is the full list of augmented assignment operators; the effects of these operators are the same as for the base operators listed in Table 3-4:

```
+= -= *= /= %= **= <<= >>= &= ^= |=
```

Operation	Description
x << y	Left shift (moves the binary form of x, y digits to the left), for example, 1 << 2 = 4
x >> y	Right shift (moves the binary form of x, y digits to the right), for example, 16 >> 2 = 4
x & y	Bitwise and
x | y	Bitwise or
x ^ y	Bitwise exclusive or (xor)
~x	Bitwise negation

Table 3-4. *Bitwise/Shift Operators for Integer Numbers.*

Numeric Functions

In addition to the operators mentioned in Table 3-2 earlier in this chapter, Python also has a small set of built-in functions that operate directly on numerical objects. See Table 3-5 for a list of these functions. Note that this list does not cover those functions that convert objects between different types; the section "Type Conversion" later in this chapter explains those functions.

There are more numeric functions and some common constants defined in the **math** module; they are described in Chapter 10.

Function	Description
abs(x)	Returns the absolute (numerical) value of a number, ignoring any signage. If the **x** is a complex number, the magnitude is returned.
coerce(x,y)	Translate the two numbers **x** and **y** into a common type, using the normal expression rules, returning the two numbers as a tuple. For example, the statement **coerce(2,3.5)** returns **(2.0,3.5)**.
divmod(x, y)	Divides **x** by **y** returning a tuple containing the quotient and remainder as derived by long division. This function is effectively equivalent to **(a/b, a%b)**.
pow(x, y [, z])	Raises **x** to the power of **y**. Note that the return value type is the same as the type of **x**. This means that you can't raise an integer to the negative power or raise any number to a power beyond the range of the object type. For example, the statements **pow(2,-1)** and **pow(2,250)** both raise exceptions. Similarly, the expression **pow(256,(1/2))** returns 1 because the **1/2** calculation rounds down to 0. If the argument **z** is added, the return value is **pow(x,y)%z**, but it's calculated more efficiently.
round(x [, y])	Round the floating-point number **x** to 0 digits, or to **y** digits after the decimal point if **y** is specified. Note that the number returned is still a floating-point number. Use **int** to convert a floating-point number to an integer, but note that no rounding takes place.

Table 3-5. *Numeric Functions*

Strings

Strings in Python work differently from those in other scripting languages. Python strings operate in the same basic fashion as C character arrays—a string is a sequence of single characters.

The term *sequence* is important here because Python gives special capabilities to objects that are based on sequences. Other sequence objects include lists, which are sequences of objects, and tuples, which are immutable sequences of objects. Strings are also immutable; that is, they cannot be changed in place. You'll see what that really means shortly. Python strings are also your first introduction to the complex objects that Python supports, and they form the basis of many of the other object types that Python supports.

String constants are defined using single or double quotes:

```
string = 'Hello World!'
string = "Hi there! You're looking great today!"
```

Note the two formats here—there is no difference within Python between using single or double quotes in string constants. By allowing both formats, you can include single quotes in double-quoted constants and double quotes in single-quoted constants without the need to escape individual characters.

Python also supports triple-quoted blocks:

```
helptext = """This is the helptext for an application
that I haven't written yet. It's highly likely that the
help text will incorporate some form of instruction as
to how to use the application which I haven't yet
written. Still, it's good to be prepared!"""
```

Python continues to add the text to the string until it sees three triple quotes. You can use three single or three double quotes—just make sure you use the same number of quotes to start and terminate the constant! Also, you don't have to follow the normal rules regarding multiline statements; Python incorporates the end-of-line characters in the text unless you use the backslash to join the lines. The preceding example, when printed, is split onto the same five original lines. To create a single, unbroken paragraph of text, you need to append a backslash to the end of each line, as follows:

```
helptext = """This is the helptext for an application \
that I haven't written yet. It's highly likely that the \
help text will incorporate some form of instruction as \
to how to use the application which I haven't yet \
written. Still, it's good to be prepared!"""
```

Putting more than one quoted constant on a line results in the constants being concatenated, such that

```
string = 'I' "am" 'the' "walrus"
print string
```

creates the output "Iamthewalrus". Note the lack of spaces.

You can also concatenate string objects and/or constants using the + operator, just as you would with numbers:

```
greeting = 'Hello '
name = 'Martin'
print greeting + name
```

Note, however, that you cannot concatenate strings with other objects. The expression

```
print 'Johnny ' + 5
```

raises an exception because the number object is not automatically translated to a string. There are ways around this—you'll see some examples shortly.

Strings can also be multiplied (repeated) using the * operator. For example, the expression

```
'Coo coo ca choo' * 5
```

produces the string

```
'Coo coo ca chooCoo coo ca chooCoo coo ca chooCoo coo ca chooCoo
coo ca choo'
```

Finally, Python provides the built-in **len** function which returns the number of characters in a string. The expression

```
len('I am the walrus')
```

returns the value 15.

The **len** function actually calculates the length of any object that has a size (lists, dictionaries, etc.).

Strings Are Just Arrays

Strings are, as you've already seen, sequences. This means that you can access the individual characters in a string using array notation, and you can access strings as if they were lists.

The array notation for a string follows the same basic format as for other languages; you append square brackets to the variable name, such that:

```
string = 'I returned a bag of groceries'
print string[0]
```

displays the first character of the string.

Like C, the offset starts at 0 as the first character or element of the string. But unlike C, the indexing also allows you to specify a range, called *slicing*, and Python is able to determine the position in the string from the end, rather than from the beginning of the string. For example, the script

```
string = 'I returned a bag of groceries'
print string[0]
print string[2:10]
print string[-1]
print string[13:]
print string[-9:]
print string[:-9]
```

creates the following output:

```
I
returned
s
bag of groceries
groceries
I returned a bag of
```

The format of a slice is:

```
string[start:end]
```

The processes of indexing and slicing use the following rules:

- The returned string contains all of the characters starting from **start** up until but not including **end**.
- If **start** is specified but **end** is not, the slice continues until the end of the string.

- If **end** is specified but **start** is not, the slice starts from 0 up to but not including **end**.
- If either **start** or **end** are specified as a negative number, the index specification is taken from the end, rather than from the start of the string where -1 is the last character.

These rules make more sense when used in combination with the diagram shown in Figure 3-1.

You should now be able to determine that the statement

```
print string[0]
```

prints the first character,

```
print string[2:10]
```

prints characters 3 through 9,

```
print string[-1]
```

prints the last character,

```
print string[13:]
```

prints all of the characters after the fourteenth character,

```
print string[-9:]
```

prints the last nine characters, and

```
print string[:-9]
```

prints everything except for the last nine characters.

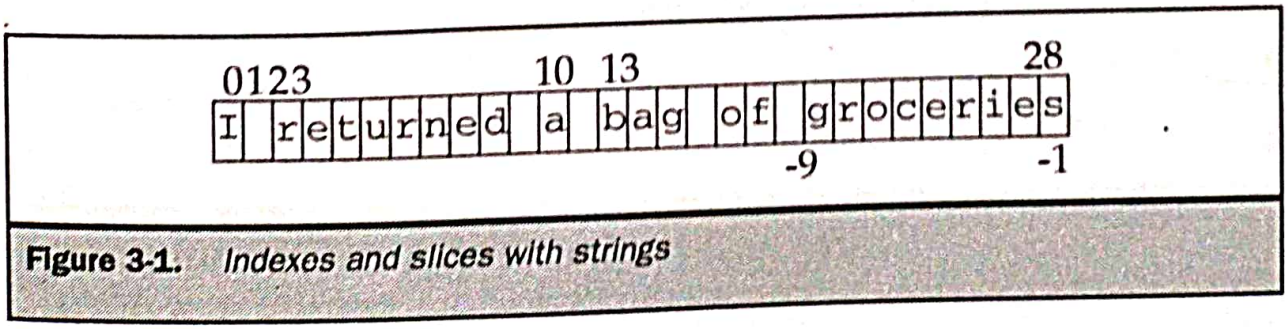


Figure 3-1. *Indexes and slices with strings*

Although Python allows you to perform slices and other operations on strings, you must remember that strings are immutable sequences—they cannot be changed in place. Therefore, the statement

```
string[:-9] = 'toffees'
```

raises an exception. The only way to change the contents of a string is to actually make a new one. This means that for apparently simple operations such as the one just attempted, you'd have to use the following sequence:

```
newstring = string[:-9] + 'toffees'
string = newstring
```

There are, of course, easier ways to change the contents of a string. One way is to use a function supplied in an external module (**string**) to do the work for us. The other way is to use string formatting, which is discussed in the next section.

Formatting

The % operator in Python allows you to format strings and other objects using the same basic notation as supported by the **sprintf** function in C. Because you're using an operator and not a function, the usage is slightly different. You put the format string on the left of the % operator, and the object (or tuple of objects) that you want interpolated into the returned expression on the right. For example, the statement

```
'And the sum was $%d, %s' % (1, 'which wasn't nice')
```

returns

```
And the sum was $1, which wasn't nice
```

The return value is always a string. The % operator accepts the same list of options as the C **sprintf** function; see Table 3-6 for a complete list of conversion formats.

Python also supports flags that optionally adjust the output format. These are specified between the % and conversion letter, as shown in Table 3-7.

The % operator also works with dictionaries by quoting the name of the dictionary element to be extracted using the format %(NAME) followed by the formatting codes listed in Tables 3-6 and 3-7. For example, the statements

```
album = {'title':'Flood', 'id':56}
print "Catalog Number %(id)05d is %(title)s" % album
```

produce the string "Catalog Number 00056 is Flood".

Format	Result
%%	A percent sign.
%c	A character with the given ASCII code.
%s	A string. Python in fact converts the corresponding object to a string before printing, so %s can be used for any object. (See the section "Type Conversion" for more details.)
%d	A signed integer (decimal).
%u	An unsigned integer (decimal).
%o	An unsigned integer (octal).
%x	An unsigned integer (hexadecimal).
%X	An unsigned integer (hexadecimal using uppercase characters).
%e	A floating-point number (scientific notation).
%E	A floating-point number (scientific notation using E in place of e).
%f	A floating-point number (fixed decimal notation).
%g	A floating-point number (%e of %f notation according to value size).
%G	A floating-point number (as %g, but using E in place of e when appropriate).
%p	A pointer (prints the memory address of the value in hexadecimal).
%n	Stores the number of characters output so far into the next variable in the parameter list.

Table 3-6. *Conversion Formats for the % Operator*

Flag	Result
space	Prefix positive number with a space.
+	Prefix positive number with a plus sign.
-	Left justify within field.
0	Use 0s, not spaces, to right justify.
#	Prefix non-zero octal with 0 and hexadecimal with 0x.
number	Minimum field width.
.number	Specify precision (number of digits after decimal point) for floating-point numbers.

Table 3-7. *Optional Formatting Flags*

Escape Characters

In string constants, you already know that you can embed a single or double quote, provided that you use double or single quotes respectively to define the entire string. With a triple-quoted block of text, you can use either single or double quotes directly. For those situations where you need to insert a quote or other special character into a string, Python supports the backslash (\) escape character. Table 3-8 lists Python's supported escape sequences.

Escape Character	Description
\ (at end of line)	Continuation (appends next line before parsing).
\\	Backslash.
\'	Single quote.
\"	Double quote.
\a	Bell.
\b	Backspace.
\e	Escape.
\000	Null. Python strings are not null terminated.
\n	Newline or linefeed.
\v	Vertical tab.
\t	Horizontal tab.
\r	Carriage return.
\f	Formfeed.
\0yy	Character represented by the octal number yy. (For example, \012 is equivalent to a newline.)
\xyy	Character represented by the hexadecimal number yy. (For example, \x0a is equivalent to a newline).
\y	Any other character y not listed above is output as normal.

Table 3-8. *Python's Supported Escape Character Sequences*

Raw Strings

For raw strings—those situations where you do not want any escape character processing—you can use the **r''** and **R''** raw strings. For example, the statement

```
print r'\a\n\x99'
```

actually outputs the string "\a\n\x99".

Raw strings are used primarily for regular expressions, where the backslash character is required for escaping regular expression operations. Chapter 10 discusses raw strings and regular expressions in more detail.

Other String Operations

Python has some facilities for sequences that are not included here. These are the iteration, membership, and **min** and **max** functions. Although these functions can be readily used on any type of sequence, including strings, they often make more sense when applied to lists or tuples. See the section on "Working with Sequences" later in this chapter for more details.

Lists

Lists are another form of sequence object and therefore they inherit many of the operational parameters of strings; after all, a string is just list of characters. However, unlike strings, Python lists can contain a list of any objects. You can store numbers, strings, other lists, dictionaries, and indeed any other object type that you create within a list, and even within the same list. Because it is a list of objects—and because all Python data is stored as an object—the list can be made up of any combination of information you choose.

Python stores a list of objects (or rather names that refer to objects), rather than a list of strings or numbers. This allows the Python list to be used in any situation where you want a list of information but don't want to be restricted to the type of information that you can store. See the section "What are Objects?" later in this chapter for more information how Python stores objects. .

Using Lists

Python creates a list when you enclose a series of objects or constants within a pair of square brackets, as in the following statements:

```
list = [1, 2, 3, 4]
songs = ['I should be allowed to think', 'Birdhouse in your Soul']
```

The use of the square brackets automatically implies a list of objects. Perl users take note; using parentheses instead of square brackets results in a tuple, described later in this chapter. You can also nest lists like the following one, which is actually a list containing two lists:

```
hex = [[0,1,2,3,4,5,6,7,8,9], ['A', 'B', 'C', 'D', 'E', 'F' ]]
```

Like strings, lists are referenced according to the index of the item you are looking for—indexes start at 0. Using the preceding examples, the first song name could be extracted by the following statement:

```
print songs[0]
```

To access an entire nested list, specify the index for the list object:

```
print 'Numbers:', hex[0]
```

To access a specific object within a nested list, use the following technique:

```
print 'D:', hex[1][3]
```

Also, note that the same slice operations work:

```
print 'A-C', hex[1][0:3]
```

To get the length of a list, use the **len** function:

```
print len( hex )
```

You can concatenate lists by using the + operator, which works in the same fashion as with string and numeric objects:

```
>>>hex[1] + list
['A', 'B', 'C', 'D', 'E', 'F', 0, 1, 2 ,3]
```

In addition, you can use the augmented addition assignment to add items to the list, but you must specify a list as the object to be added, as in the following example:

```
>>> songs += ['AKA Driver']
```

Finally, you can multiply lists by a numeric object to repeat the elements within a list:

```
>>>list * 2
[0, 1, 2, 3, 0, 1, 2, 3]
```

Note though that you cannot multiple a list by a list—not that it would make much sense if you could!

However, you can create a new list based on a simple expression. For example, given the list

```
>>> list = [1,2,3,4]
```

If you want a list of the cubes of each element, you could use the **map** function (described in Chapter 9) in combination with an anonymous **lambda** function (described in Chapter 4):

```
>>> cubes = map(lambda(x): x**3, list)
```

With Python 2.0 onwards, you can simply rewrite the preceding statement as follows:

```
>>> cubes = [x**3 for x in list]
```

Lists are Mutable

Remember how we said strings were immutable objects but lists were mutable? Well, the significance is that you can modify a list in place. With a string, the following operations raise an exception:

```
string = 'Narrow your Eyes'
string[7:11] = 'her'
```

However, with a list, the following operations work:

```
list = [0, 1, 2, 4, 16]
list[3:4] = [4, 8]
```

and the result is a list with six elements, **[0, 1, 2, 4, 8, 16]**. Note that you have to specify a slice; if you tried using **list[3]** in the second line in the preceding example, you'd replace

the fourth element with an embedded two-element list. Also, the number of elements you're replacing doesn't have to match the number of elements you're inserting.

To delete items from the list, you need to use the **del** function, which accepts an element or slice from a list to be deleted, such that:

```
del list[3]
```

deletes the fourth element and

```
del list[1:4]
```

deletes the middle three elements from the list.

List Methods

Programmers familiar with objects know that many objects come with their own set of methods. Methods are special functions that are part of the object type definition. Methods are in fact just functions that happen to be specific to a particular type of object. Lists support a number of default methods that control the list contents. To use a method, you must qualify the method with an object name, for example, to call the **sort** method:

```
numbers = [3, 5, 2, 0, 4]
numbers.sort()
```

Note that with most methods, the new list is not returned; the list is modified in place. In the following example, the **numbers** object is now sorted. This complicates operations when you want to output a sorted list because the following statement will not print what you expect:

```
print numbers.sort()
```

Instead, you must do it in two lines:

```
numbers.sort()
print numbers
```

On the other hand, performing operations on the list makes it clear that we are modify a list, and that subsequent accesses will be on a sorted version of the list.

Table 3-9 lists the methods supported for lists.

Method	Description
append(x)	Appends the single object x to the end of the list. Supplying multiple arguments raises an exception. To append a tuple, explicitly supply a tuple, for example, **list.append((1,2,3))**. Returns **None**.
count(x)	Returns a count of the number of times the object x appears in the list.
extend(L)	Adds the items in the list L to the list. Returns **None**.
index(x)	Returns the index for the first item in the list matching the object x. Raises an exception if there is no matching element.
insert(i, x)	Inserts the object x before the element pointed to be i. Thus, **list.insert(0,x)** inserts the object before the first item. Returns **None**.
pop(x)	Removes the item at point x within the list and returns its value. If no index is specified, **pop** returns the last item in the list.
remove(x)	Deletes the first element from the list that matches the object x. Raises an exception if there is no matching element. Returns **None**.
reverse()	Reverses the order of the elements in the list, in place. Returns **None**.
sort()	Sorts the list in place. Returns **None**.

Table 3-9. *Methods Supported by List Objects*

Here are some examples using the methods described in Table 3-9:

```
list = [1,2,3]
more = [11,12,13]
list.append(4)            # [1,2,3,4]
list.append(('5a','5b'))# [1,2,3,4,(5a,5b)]
list.extend(more)         # [1,2,3,4,(5a,5b),11,12,13]
list.index(11)            # Returns 5
list.insert(5,'Six')      # [1,2,3,('5a','5b'),'Six',11,12,13]
list.pop()                # Returns (and removes) 13
list.pop(4)               # Returns the tuple ('5a','5b')
list.remove('Six')        # [1,2,3,11,12]
list.reverse()            # [12,11,3,2,1]
list.sort()               # [1,2,3,11,12]
```

You haven't come across the **None** value before, although it can apply to any object. The value **None** really means nothing—it does not imply an empty list or a zero-length string; instead, it implies an object that has no value. The **None** value is important because it can be used to identify an unpopulated object, similar in a sense to the C NULL value and identical in essence to the Perl **undef**.

Other List Operations

Because lists are another type of sequence, there are some additional features of list objects that have not been covered in this section. Please see the section "Working with Sequences" later in this chapter for details.

Tuples

Tuples are, to all intents and purposes, identical to lists except for one detail: tuples are immutable. Unlike lists, which you can chop and change and modify as you like, a tuple is a nonmodifiable list. Once you have created a new tuple, it cannot be modified without making a new tuple.

To create a tuple instead of a list, use parentheses instead of square brackets when defining the tuple contents, as in the following example:

```
months = ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
          'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec')
```

The preceding statement is actually a very good example of why Python has tuples. Because they cannot be modified, you can use them to store information that you do not want to be altered. In the case of the preceding statement, you are interested in the order of the elements. If you had created this object as a list instead of as a tuple and then accidentally sorted it, the list of months would be largely useless. Without the ordered sequence, the information has no value.

Note that if you try to modify a tuple, an exception is raised to notify you of the error. This can be useful during development and debugging because it helps spot instances where you are misusing an object type.

The following are all examples of tuple use:

```
one = (1,)
four = (1, 2, 3, 4)
five = 1, 2, 3, 4, 5
nest = ('Hex', (0,1,2,3,4,5,6,7,8,9), ('A', 'B', 'C', 'D', 'E', 'F'))
```

The first and third lines in the preceding statements are important to understand. In the case of the first line, to create a one-element tuple you must use the redundant-

looking comma; otherwise, the Python interpreter parses (1) as a constant expression—the inclusion of the comma forces Python to interpret the statement as a tuple.

The third line shows tuple creation without the use of parentheses. Python allows you to create a tuple without using parentheses in those situations where the lack of parentheses is not ambiguous. In this instance, it's obvious you are creating a tuple. As a general rule, you should use parentheses to highlight to you and any other programmers looking at your code that you are introducing a tuple.

Using Tuples

Tuples support the same basic operations as lists. You can index, slice, concatenate, and repeat tuples just as you can lists. The **len** function also works since a tuple is a sequence. The only actions you can't perform with a tuple are to modify them or to perform any of the methods available to lists since these methods make modifications.

It's also worth noting again that tuple operations return tuples. The statement

```
three = five[:3]
```

makes **three** a tuple, not a list.

Some functions also return tuples rather than lists, and some functions even require tuples as arguments. See "Type Conversions" for information on translating between Python object types.

Working with Sequences

The object types of strings, lists, and tuples belong to a generic Python object class called *sequences*. A sequence object is any object whose elements can be accessed using a numbered index. In the case of a string, each character is available via the index; for lists, it's each object. Because sequences have an order and are made up of one or more elements, there are occasions when you want to use the list as either a sequence or a traditional list of values to use as a reference point.

Python provides a single operator, **in**, that allows you to access the contents of a list in a more logical fashion.

Membership

If you are working with a list of possible values for a particular structure and want to verify the existence of a single element within the list, there are normally two choices available to you. Either you write a function to work through the contents of the list looking for the object, or you use a superior data structure such as a dictionary to store the item and then use the object's built-in methods to determine whether the object is a member of your pseudo-list.

Within Python, you can use the **in** operator to determine whether a single object is a member of the list, returning 1 if the element is found, as in the following example:

```
>>> list = [1, 2, 3]
>>> 1 in list
1
>>> 4 in list
0
```

This process also works with strings:

```
>>> word = 'supercalifragilisticexpialidocious'
>>> 'x' in word
1
```

and tuples:

```
>>> days = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun')
>>> 'Mon' in days
1
```

Iteration

You haven't yet learned about the loop operators supported by Python. Most of the usual operations of **while** and **for** are supported, and we can also iterate through the individual elements of a sequence. Again, Python uses the **in** operator, which this time returns each element of the sequence to each iteration of the loop:

```
for day in days:
   print day
```

You'll see more examples of this later in this chapter and throughout the rest of the book.

Dictionaries

A dictionary is an associative array; instead of using numbers to refer to individual elements of the list, you use other objects—generally strings. A dictionary element is split into two parts, the *key* and the *value*. You access the values from the dictionary by specifying the key.

Like other objects and constants, Python uses a strict format when creating dictionaries:

```
monthdays = {'Jan' : 31, 'Feb' : 28, 'Mar' : 31,
             'Apr' : 30, 'May' : 31, 'Jun' : 30,
             'Jul' : 31, 'Aug' : 31, 'Sep' : 30,
             'Oct' : 31, 'Nov' : 30, 'Dec' : 31}
```

You can also nest dictionaries, as in the following example:

```
albums = {'Flood' : {1:'Birdhouse in your Soul'}}
```

To access an entry you use the square brackets to define the index entry of the element you want to be returned, just as you do with strings or lists:

```
print 'January has',monthdays['Jan'],'days'
print 'First track is',albums['Flood'][1]
```

However, the interpreter raises an exception if the specified key cannot be found. See the section "Dictionary Methods" later in this chapter for details on the **get** and **has_key** methods. Again, just like lists, you can modify an existing key/value pair by redefining the entry in place:

```
monthdays['Feb'] = 29
```

If you had specified a different key, you would have added, rather than updated the entry. The following example

```
monthdays['feb'] = 29
```

introduces a thirteenth element to the dictionary.

You can delete entries in a dictionary using the **del** function:

```
del monthdays['Feb']
```

Dictionaries are not sequences, so concatenation and multiplication will not work—if you try them, both operations will raise an exception.

Using Dictionaries

Although dictionaries appear to work just like lists, the two are in fact very different. Dictionaries have no order—like lists, the indexes (keys) must be unique—but there is no logical order to the keys in the dictionary. This means that a dictionary is not a sequence; you cannot access the entries in a dictionary sequentially in the same way you can access entries in a string, list, or tuple. This also has the unpleasant side effect

that when accessing the same dictionary at two separate times, the order in which the information is returned by the dictionary does not remain constant.

However, the dictionary object provides two methods, **keys** and **values**, that return a list of all the keys and values defined within the dictionary. For example, to get a list of months from the dictionary of months in the earlier example, use the following statement:

```
months = monthdays.keys()
```

The **months** object is now a list of the month strings that formed the keys of the **monthdays** dictionary.

The **len** function returns the number of elements, the key/value pairs, stored in a dictionary, such that the statement

```
len (monthdays)
```

returns the correct value of 12.

Dictionary Methods

Table 3-10 lists the full list of the methods supported by the dictionary object type.

Method	Description
has_key(x)	Returns true if the dictionary has the key **x**.
keys()	Returns a list of keys.
values()	Returns a list of values.
dict.items()	Returns a list of tuples; each tuple consists of a key and its corresponding value from the dictionary **dict**.
clear()	Removes all the items from the dictionary.
copy()	Returns a copy of the top level of the dictionary, but does not copy nested structures; only copies the references to those structures. See the section "Nesting Objects."
update(x)	Updates the contents of the dictionary with the key/value pairs from the dictionary **x**. Note that the dictionaries are merged, not concatenated, since you can't have duplicate keys in a dictionary.
get(x [, y])	Returns the key **x** or **None** if the key cannot be found, and can therefore be used in place of **dict[x]**. If **y** is supplied, this method returns that value if **x** is not found.

Table 3-10. *Methods for Dictionaries*

Sorting Dictionaries

The **keys()** and **values()** methods to a dictionary object return a list of all the keys or values within the dictionary. This is useful when iterating over a dictionary or when using a dictionary for de-duplicating values. However, we cannot nest the **keys()** or **values()** and the corresponding list or tuple **sort()** methods into one call. To get a sorted list, it's a two-stage process:

```
months = monthdays.keys()
months.sort()
```

Many users try the following:

```
months = monthdays.keys().sort()
```

But that won't work. The reason is quite simple—the **sort** method modifies a list in place; it doesn't return the new list. Although the object generated by the **keys** method is sorted, the information is never returned and when the statement ends, the temporary sorted list object is destroyed, while **months** contains the special value **None**.

The problem with this method is not really apparent until you try to list the contents of a dictionary in an ordered fashion. You have to use the two-stage method shown in the first example of sorting a dictionary:

```
keys = monthdays.keys()
keys.sort()
for key in keys:
...
```

If you want to sort the list based on the values rather than on the keys, the process gets even more complicated. You can't access the information in a dictionary using values, only using the keys. What you need to do is sort a list of tuple pairs by supplying a custom comparison function to the **sort** method of a list.

You can get a list of tuple pairs using the **items** method. The process looks something like this:

```
monthdays = {'Jan' : 31, 'Feb' : 28, 'Mar' : 31,
             'Apr' : 30, 'May' : 31, 'Jun' : 30,
             'Jul' : 31, 'Aug' : 31, 'Sep' : 30,
             'Oct' : 31, 'Nov' : 30, 'Dec' : 31}
months = monthdays.items()
months.sort(lambda f, s: cmp(f[1], s[1]))
for month, days in months:
    print 'There are',days,'days in',month
```

We haven't covered many of the techniques used in the above example, but here's how it works. Line 5 in the preceding example gets a list of the key/value pairs as tuples. Line 6 sorts the list by comparing the value component of each tuple; the **lambda** is an anonymous function. (See Chapter 4 for more information on the **lambda** function and see Chapter 7 for an explanation of the custom element to the **sort** method.) Then the **for** loop extracts the month and number of days in each individual tuple from the list of tuples before printing each one.

When executed, the script output looks like this:

```
There are 28 days in Feb
There are 30 days in Jun
There are 30 days in Nov
There are 30 days in Apr
There are 30 days in Sep
There are 31 days in Aug
There are 31 days in May
There are 31 days in Oct
There are 31 days in Jul
There are 31 days in Jan
There are 31 days in Dec
There are 31 days in Mar
```

Dictionaries are one of the most useful storage elements because they enable you to access information using anything as a key—Python allows you to use any object as a key to the key/value pair. Chapter 10 takes a closer look at dictionaries in Python.

Files

Python's access to files is built into Python in the same way as the other object types you've already seen. You create a file object by using the built-in **open** function, and then you use methods on the new object to read and write information to and from the file. This eliminates a lot of the complexity of the file handle model used by C, Perl, and VB; instead, you get a consistent interface to what is essentially just another form of storage available on your machine.

We won't look at the specifics of using files for the moment; we'll leave that to Chapter 11 when you'll learn how to use Python for processing and managing files. However, to get a taste, look at the following script that opens and reads each line from the file **myfile.txt** before displaying the lines to the screen:

```
input = open('myfile.txt')
for line in input.readlines():
    print line
input.close()
```

Object Storage

Although Python always creates objects when you create a variable, it's important to understand the association between the name, the object, and the variable. At the risk of sounding like a stuck record, we'll reiterate again the statement we made at the beginning of this chapter: Python stores information as objects. In truth, Python actually stores references or pointers to the internal structures used to store the information for a specific object type. This is the equivalent of the C pointer or the Perl reference—a Python object is merely the address of a structure stored in memory.

With C, to introduce a special variable into your application you have to use either a pointer to a structure or predefine a structure before you use the data type. Using pointers, while practical at a system level, is difficult and adds extra layers of complexity that you don't need. In addition, the programmer needs to be aware that the variable they're dealing with is a pointer and therefore requires special treatment.

With Perl, the programmer has to make a conscious effort to use a reference as a specific data type. You must dereference a Perl reference before you can access the information. If you don't dereference or you dereference to the wrong type (i.e., try to access a hash as a scalar), you get either the physical address of the variable or an error during compilation. This problem is easy to spot, but it adds extra complication to the programming process that frankly you could do without.

Unlike Perl and C however, the Python programmer never needs to be aware of the fact that the pointers exist. When you access an object, you are accessing the information stored in it, irrespective of how the object is defined or where or how the object information is actually stored.

Objects, Names, and Variables

The Python *name* is the alphanumeric name given to the pointer that points to the object. Note the following example:

```
greeting = 'Hello World'
```

In the preceding statement, the name is **greeting**. The object has no identifiable name; it's just an allocated patch of memory which is pointed to by **greeting**. But the object's type is a string. The combination of the two can be thought of as a typical *variable*.

The significance of the pointer is important. Consider what happens when you reassign the "value" of the variable:

```
greeting = ['Hello', 'World']
```

The name has remained the same, but the object to which it points is in a different physical location within memory, and is of a completely different type.

Variable Name Rules

Python follows some very basic rules for naming variables:

- Variables must begin with an underscore or a letter, and they may contain any combination of letters, digits, or underscores. You cannot use any other punctuation characters.
- Variables are case sensitive—**string** and **String** are two different variable names.
- Reserved words cannot be superceded by variables. The full list of reserved words is as follows:

and	**assert**	**break**	**class**	**continue**
def	**del**	**elif**	**else**	**except**
exec	**finally**	**for**	**from**	**global**
if	**import**	**in**	**is**	**lambda**
not	**or**	**pass**	**print**	**raise**
return	**try**	**while**		

Because Python is case sensitive, there is no reason why you cannot have variables called **Return** and **RETURN**, although it's probably a bad idea.

Python Copies References, Not Data

When you refer to a Python object by name when assigning information to an object, Python stores the reference, not the data. For example, the following code appears to create two objects:

```
objecta = [1,2,3]
objectb = objecta
```

In fact, you have only created one object but you have created two names (pointers) to that object, as demonstrated by trying the statements interactively:

```
>>> objecta = [1,2,3]
>>> objectb = objecta
>>> objecta[1] = 4
>>> print objectb
[1, 4, 3]
```

To create a copy of an object, you must force Python to explicitly assign either the value of the object or you can use one of the type converters described shortly. The former method is often the easiest with lists and tuples:

```
objectb = objecta[:]
```

The effects of copying the pointer to the object rather than copying the contents of the object pose some interesting possibilities. Most useful of all is the fact that because this is Python's normal behavior, building complex structures becomes a natural rather than a forced process. For example, you can use objects to specify the keys used for dictionaries. You can build a list of tables for your **contacts** database like this:

```
tables = {contacts : 'A list of contacts',
          addresses : 'The address list'}
```

It doesn't matter if the contents of **contacts** change because the key specified in the **tables** dictionary is just a pointer to the **contacts** object; it's not a copy of the **contacts** dictionary.

Nesting

Because Python copies object pointers, you can use objects anywhere; you are not limited to the normal boundaries and restrictions placed on variables in other languages. For example, you have already seen that a list can contain a mixture of strings and numbers and even other lists. For example, the following structure describes a list of contacts using a combination of lists and dictionaries to describe the information:

```
contacts = [{'Name' : 'Martin',
             'Email' : 'mc@mcwords.com'},
            {'Name' : 'Bob',
             'Email' : 'bob@bob.com'}]
```

Part 3 of this book contains more examples of nesting as you start to learn in more detail how you can use Python to solve specific problems.

Type Conversion

Beyond the methods and operators already discussed for the individual object types, there are a number of built-in functions that can convert data from one type to another. Table 3-11 contains the full list of such functions.

There is also a special operator supported by Python that works in the same way as the **str()** built-in function. The `` (back ticks) operator evaluates the enclosed statement and then returns a string representation, as in the following example:

```
>>> print `(34*56.0)`
1904
>>> album = ('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul')
>>> album
('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul')
>>> `album[:2]`
```

```
"('TMBG', 'Flood')"
>>> str(album[:2])
"('TMBG', 'Flood')"
>>> `list(album)`
"['TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul']"
```

Note that the format used by both **str** and `` when returning the information is the same as would be required to build the object. Therefore, to display the Python statements required to build the earlier album list, use the following statements:

```
contacts = [{'Name' : 'Martin',
             'Email' : 'mc@mcwords.com'},
            {'Name' : 'Bob',
             'Email' : 'bob@bob.com'}]
>>> >>> `contacts`
"[{'Email': 'mc@mcwords.com', 'Name': 'Martin'}, {'Email': 'bob@bob.com',
'Name': 'Bob'}]"
```

Function	Description
str(x)	Translates the object x into a string.
list(x)	Returns the sequence object x as a list. For example, the string "hello" is returned as ['h', 'e', 'l', 'l', 'o']. Converts tuples and lists to lists.
tuple(x)	Returns the sequence object x as a tuple.
int(x)	Converts a string or number to an integer. Note that floating-point numbers are truncated, not rounded; **int(3.6)** becomes 3.
long(x)	Converts a string or number to a long integer. Conversion as for **int**.
float(x)	Converts a string or number to a floating-point object.
complex(x,y)	Creates a complex number with real part of x and imaginary part of y.
hex(x)	Converts an integer or long to a hexadecimal string.
oct(x)	Converts an integer or long to an octal string.
ord(x)	Returns the ASCII value for the character x.
chr(x)	Returns the character (as a string) for the ASCII code x.
min(x [, ...])	Returns the smallest element of a sequence.
max(x [, ...])	Returns the largest element of a sequence.

Table 3-11. *Built-in Type Conversions*

Type Comparisons

When you compare two objects within Python, the interpreter compares the values of both objects before returning the result of the comparison. This means that when you compare two lists, the contents of both lists are examined to determine whether they are identical. The same is true of all other object tests—the interpreter checks the contents of each data structure when making the comparison.

Table 3-12 lists the supported operators for type comparisons.

There are some special cases shown in Table 3-12. If you create two identical lists and compare their values, you should get a return value of 1, to indicate that the two lists are of equal value:

```
>>> lista = [1, 2, 3]
>>> listb = [1, 2, 3]
>>> lista == list
1
```

Note as well that the order as well as the values is important:

```
>>> listb = [2, 3, 1]
>>> lista == listb
0
```

Operator	Description
x < y	Less than.
x <= y	Less than or equal to.
x > y	Greater than.
x >= y	Greater than or equal to.
x == y	Have equal value.
x != y, x <> y	Do not have equal value.
x is y	Pointers to same object.
x is not y	Different objects.
not y	Inverse—returns true if x is false/false if x is true.
x or y	Returns x if x is true, or y if x is false.
x and y	Returns x if x is false, or y if x is true.
x < y < z	Chained comparisons; returns true only if all operators return true.

Table 3-12. *Comparison Operators*

To compare whether the two objects are identical, e.g., that they both point to the same physical object, you need to compare the two objects using the **is** operator:

```
>>> lista is listb
0
```

You need to use the **is** operator because there are two separate objects. Copying the object reference from **lista** to **listb** however

```
>>> listb = lista
>>> lista is listb
1
```

gives us the result you expect.

The **is** operator is useful when you want to check the source and validity of a object against a source or control object.

Python makes its comparisons using the following rules:

- Numbers are compared by magnitude.
- Strings are compared character by character.
- Lists and tuples are compared by element, from the lowest to highest index.
- Dictionaries are compared by comparing sorted key/value pairs.

All comparisons within Python return 1 if the result is true, or 0 if the comparison fails. In addition, Python treats any non-zero value as true, including a string. The only exceptions to this rule are **None**, which is false, and any empty object (list, tuple, or sequence). Table 3-13 contains a summary of the different true/false values.

Object/Constant	Value
''	False
'string'	True
0	False
>1	True
<-1	True
() (empty tuple)	False
[] (empty list)	False
{} (empty dictionary)	False
None	False

Table 3-13. *True/False Values of Objects and Constants*

Statements

The statement is the most basic form of executable element within a Python program. A variable on its own does nothing—it must be part of a statement in order to be created, modified, and manipulated. There are a number of generic statement types that you'll learn about in this section, including basic statements, assignments, function calls, and the control and loop statements.

Statement Format

Python uses a very simple method for parsing the individual statements that make up a typical program. Within Python, each line is identified as a single statement; the normal line termination of a carriage-return or linefeed acts as the statement terminator. For example, the following two-line program is valid:

```
print "Hello World!"
print "I am a test program"
```

For extra long lines, such as the following line, you have a number of options, depending on the line contents:

```
print "Hello, I am a test program and I am printing an extra long
line so I can demonstrate how to split me up"
```

For most lines, the easier method is to append a backslash to the end of the line where you want to split it, as in the following example:

```
print "Hello, I am a test program and I am printing an extra \
long line so I can demonstrate how to split me up"
```

When the Python interpreter sees the backslash as the last character in a line it automatically appends the next line before the entire line is parsed. This process is cyclic, so a line can span as many lines as you like, providing that each line ends in a backslash:

```
print "Hello, I am a test program \
and I am printing an extra \
long line so I can demonstrate \
how to split me up"
```

For statements that incorporate a pair of matching parentheses, you do not have to use the backslash technique. Python automatically searches the next line looking for the terminating parenthesis. You can modify your **print** statement in the preceding

example to print a tuple of the message, which requires parentheses and therefore implies a termination character:

```
print ("Hello, I am a test program ",
"and I am printing an extra ",
"long line so I can demonstrate ",
"how to split me up")
```

Finally, there are situations where without either the backslash or parentheses, Python automatically expects further information and therefore examines the next line.

For C and Perl programmers, Python also accepts the semicolon as a line terminator if it makes you feel more comfortable. However, the use of a semicolon is completely optional and using one has no effect on how Python parses lines—you must continue to use one of the tricks described earlier whether you use semicolons or not.

Comments

Python allows you to incorporate comments by inserting the hash symbol (#) into code as in the following example:

```
version = 1.0 # This is the current version number
```

Everything after the hash sign is taken to be a comment and is ignored by the Python interpreter. Note that this does affect hash signs embedded within quotes, as you might expect.

Assignments

The assignment is the most basic statement in Python, assigning data and an object type to a variable name. You have already seen a number of different examples of this when you learned about the different Python object types. Unlike C, but like Perl (in non-strict mode), you do not need to predeclare Python variables before you assign them a value. However, variables must exist when used within an expression or other form of statement.

There are four basic methods of assignments: basic assignments, tuple assignment, list assignment, and multiple-target assignment.

Basic Assignments

The basic assignment is the one you have seen most examples of in this chapter. A basic assignment creates a new object using the correct type according to the value

assigned to the object, and then points the specified name to the new object, as in the following examples:

```
number = 45
message = 'Hello World'
mylist = ['I Palindrome I', 'Mammal']
mytuple = ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
mydict = {'Twisting':Flood, 'Mammal':'Apollo 18'}
```

Tuple and List Assignments

You can extract the individual elements from a tuple or list using a different operator of assignment. For tuples, you can just specify the list of variables to assign to, each separated by a comma:

```
title, name = 'Mr', 'Martin'
album = ('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your Soul')
artist, title = album[0:2]
track1, track2 = album[-2:]
```

What you're in fact doing in the preceding statements is creating a new tuple on the left-hand side of the assignment operator. However, the tuple with the variable names is anonymous and the information about the entire tuple is not recorded. Because you don't have to use parentheses to indicate a tuple, the assignment looks more natural, although you could have written them as:

```
album = ('TMBG', 'Flood', 'Theme from Flood', 'Birdhouse in your
Soul')
(artist, title) = album[0:2]
(track1, track2) = album[-2:]
```

What you end up with in each case is a new variable pointing to the information on the right-hand side. Note as well that you can also use slices and indexes to extract multiple elements from a tuple.

For lists, you must create a new, anonymous list in the same way that you create an anonymous tuple:

```
album = ['TMBG', 'Lincoln', 'Ana Ng', 'Cowtown']
[artist, title] = album[0:2]
```

The tuple and list assignments have another trick up their sleeve. Because names are merely pointers, you can "swap" two object/name combinations by performing a tuple or list assignment as follows:

```
artist, title = title, artist
```

You can see this better in the following interactive session:

```
>>> title = 'Flood'
>>> artist = 'TMBG'
>>> artist, title = title, artist
>>> artist, title
('Flood', 'TMBG')
```

Multiple-Target Assignments

You can create a single object with multiple pointers using the multiple-target assignment statement:

```
group = title = 'They Might Be Giants'
```

In the preceding statement, you created a single string object with two different pointers called **group** and **title**. Remember that because the two pointers point at the same object, modifying the object contents using one name also modifies the information available via the other name. This is effectively the same as:

```
group = 'They Might Be Giants'
title = group
```

Print

Unlike other languages, the default method for communicating with the user is via a statement rather than via a function. The statement has the same basic name, **print**, but is embedded into the interpreter. Unlike similar **print** functions however, the Python **print** statement actually outputs string representations of the supplied objects to the standard output of the interpreter. The destination is the same as the C **stdout** or Perl STDOUT file handles, and cannot normally be pointed elsewhere.

Actually, you can redirect the standard output by modifying the object used to point to the standard output device. It should be no surprise that Python uses objects to communicate with the outside world. The following code demonstrates this point:

```
import sys
fp = open( 'somefile.txt', 'w' )
sys.stdout = fp
print 'this goes in the file, not on your stdout'
```

This works because instead of pointing the standard output to its originial location, you copy the object data for the new ***fp*** *object instead. The* ***stdout*** *object, which is used by the* ***print*** *function, now sends all of its output to the file* ***somefile.txt****.*

When printing, the statement follows these basic rules:

- Objects and constants separated by commas are separated by spaces when printed. Use string concatenation or the % formatting operator to avoid this.
- A linefeed is appended to every output line. To avoid this, append a comma to the end of the line.

For example, the following script, when executed, displays the different forms supported:

```
print 'Hello','World'
print 'This is a' + ' concatenated', string
print 'The first line',
print 'Plus some continuation'
print 'I ordered %d dozen %s today' % (6, 'eggs')
contacts = [{'Name' : 'Martin',
             'Email' : 'mc@mcwords.com'},
            {'Name' : 'Bob',
             'Email' : 'bob@bob.com'}]
print contacts
```

The preceding script produces the following output:

```
Hello World
This is a concatenated string
The first line Plus some continuation
I ordered 6 dozen eggs today
[{'Email': 'mc@mcwords.com', 'Name': 'Martin'}, {'Email': 'bob@bob.com',
'Name': 'Bob'}]
```

Note how in the last line, **print** displays the complex object in the same format as **str** and the `` operator.

The ability to print out the contents of an object in this way is extremely useful, both as a programming tool and for debugging. For example, you can store the configuration parameters for an application in one or more variables. When you want to save the information, just use **print** or the **str** function to write out the statements. To get the information back, all you need do is import the file contents in the same way that you would with any other module.

Control Statements

Python processes program statements in sequence until the program states otherwise through the use of a control statement or a function call. Python supports three different control statements, **if**, **for**, and **while**. The **if** statement is the most basic control statement; it selects a statement block to execute based on the result of one or more expressions. Both **for** and **while** are loop statements.

if

The **if** statement accepts an expression and then executes the specified statements if the statement returned true. The general format for the **if** statement is:

```
if EXPRESSION:
    BLOCK
elif EXPRESSION2:
    BLOCK2
else:
    BLOCK3
```

The **EXPRESSION** is the test you want to perform (see Table 3-12); a result of true means that the statements in **BLOCK** are being executed. The optional **elif** statement performs further tests when **EXPRESSION** fails, executing **BLOCK2** if **EXPRESSION2** returns true or executing **BLOCK3** otherwise. You can have multiple **elif** statements within a single **if** statement. If no expressions match, then the optional block to the **else** statement is executed.

Note that in each case, the preceding line to the statement block ends with a colon. This indicates to Python that a new statement block should be expected. Unlike many other languages, Python is a little more relaxed about its definitions of the code block. Both C and Perl use curly brackets (or braces) to define the start and end of a block. Python uses indentation—any lines after the colon must be indented (to the same level) if you want them to be in the same logical code block.

You can see how the indentation and blocks work by looking at Figure 3-2. Notice in this figure that Block 1 continues after the indentations for Block 2 and Block 3 no longer exist. See the rest of the script for details on the block definitions and indentation.

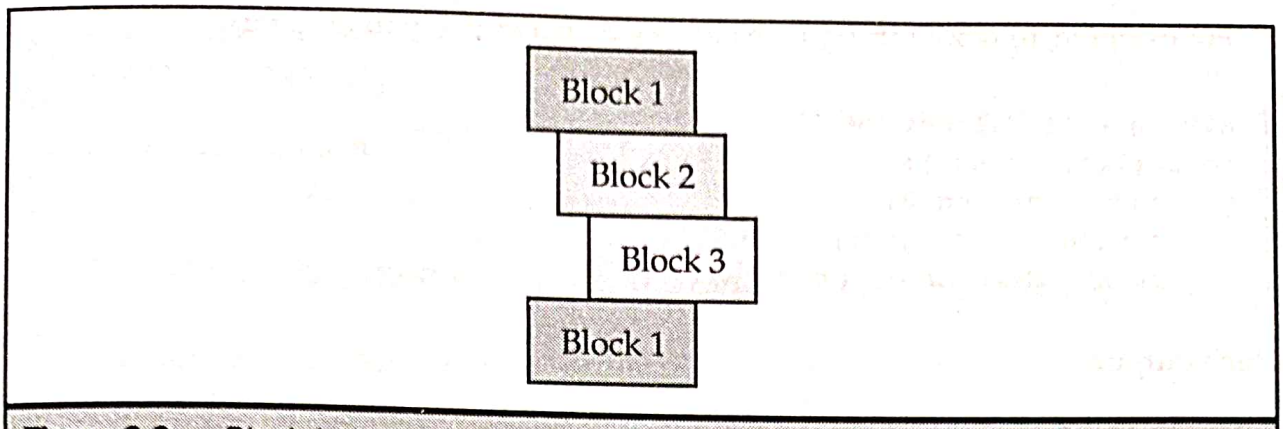


Figure 3-2. *Block indentation in Python*

Consider the following example combining **if** and **elif** statements:

```
if (result == 1):
    print 'Got a one'
elsif (result >1):
    print 'Got more than one'
else:
    print 'Got something else'
```

For simple tests, you can combine the statement and block onto a single line:

```
if (1): print 'Hello World'
```

This really only works for single-line blocks.

Note *If you want to perform a* ***switch*** *or* ***case*** *statement in Python, you must use an* ***if*** *statement with multiple* ***elif*** *expressions.*

while

The **while** loop accepts a single expression and the loop continues to iterate as long as the test continues to return true. The expression is reevaluated before each iteration. The basic format is therefore:

```
while EXPRESSION:
    BLOCK
else:
    BLOCK
```

For example, to work through the characters in a string, you might use

```
string = 'I Palindrome I'
while(len(string)):
    char = string[0]
    string = string[1:]
    print 'Give me a',char
```

which outputs

```
Give me a I
Give me a
…
Give me a
Give me a I
```

Note that in this particular example, it's probably easier to use a **for** loop.

The optional **else** block is only executed if the loop exits when **EXPRESSION** returns false, rather than the execution being broken by the **break** statement. See the example of a **for** loop in the next subsection.

for

The **for** loop is identical to the list form of the **for** loop in Perl. The basic format for the **for** loop is:

```
for TARGET in OBJECT:
    BLOCK
else:
    BLOCK
```

You specify the object to be used as the iterator within the loop and then supply any form of sequence object to iterate through. For example:

```
for number in [1,2,3,4,5,6,7,8,9]:
    print 'I can count to',number
```

For each iteration of the loop, **number** is set to the next value within the array producing the output

```
I can count to 1
I can count to 2
...
I can count to 8
I can count to 9
```

Because it works with any sequence, you can also work through strings as in

```
for letter in 'Martin':
    print 'Give me a', letter
```

which generates

```
Give me a M
Give me a a
Give me a r
Give me a t
Give me a i
Give me a n
```

In addition, just like the **while** loop, the **else** statement block is executed when the loop exits normally as in

```
for number in [1,3,5,7]:
    if number > 8:
        print "I Can't work with numbers that are higher than 8!"
        break
    print '%d squared is %d' % (number, pow(number,2))
else:
   print 'Made it!'
```

which outputs

```
1 squared is 1
3 squared is 9
5 squared is 25
7 squared is 49
Made it!
```

Ranges

Because the **for** loop does not support the loop counter format offered by Perl and C, you need to use a different method to iterate through a numerical range. There are two possible methods. Either use the **while** loop:

```
while(x <10):
    print x
    x = x+1
```

Alternatively, you can use the **range** function to generate a list of values:

```
for x in range(10):
...
```

The **range** function returns a list. The basic format of the **range** function is

```
range([start, ] stop [, step])
```

In its single-argument form, **range** returns a list consisting of each number up to, but not including, the number supplied:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If the **range** function is supplied two arguments, it returns a list of numbers starting at the first argument, up to but not including the last argument:

```
>>> range(4,9)
[4, 5, 6, 7, 8]
```

The final form, with three arguments, allows you to define the intervening step:

```
>>> range(0,10,2)
[0, 2, 4, 6, 8]
```

Because the **range** function returns a list, it can create, for very large ranges, very large lists that require vast amounts of memory to store. To get around this, use the **xrange** function. The **xrange** function has the exactly format and indeed, performs exactly the same function as **range**, but it does not create an intervening list and therefore requires less memory for very large ranges. On the other hand, it produces a

very large list that results in a processing overhead as new objects are created and added to the list structure.

Loop Control Statements

Python supports three loop control statements that modify the normal execution of a loop. They are **break**, **continue** and **pass**.

The **break** statement exits the current loop, ignoring any **else** statement, continuing execution after the last line in the loop's statement block.

The **continue** statement immediately forces the loop to proceed to the next iteration, ignoring any remaining statements in the current block. The loop statement expression is reevaluated. The example

```
x=20
while (x):
    x = x-1
    if not x % 3: continue
    print x,'is not a multiple of 3'
```

generates the following output:

```
19 is not a multiple of 3
17 is not a multiple of 3
16 is not a multiple of 3
...
4 is not a multiple of 3
2 is not a multiple of 3
1 is not a multiple of 3
```

The **pass** statement is effectively a no-op—it does nothing. The statement

```
if (1): pass
```

does nothing at all.

Although it may seem pointless, there are times when you want to be able to identify a specific event, but ignore it. This is especially true of exceptions, the Python error-handling system. You'll see **pass** and exceptions in action in Chapter 5.

Common Traps

For the programmer migrating to Python, there are a number of common traps that trick the unwary. Most relate to the very minor differences between Python and other languages, particularly if you have been used to Perl or Shellscript.

Variable Names

Variables within Python follow these basic rules and differences from other languages:

- You do not need to predeclare variables before they are used; Python sets the object type based on the value it is assigned.
- Variables are given simple alphanumeric names.
- Variables do not require qualification using special characters.
- Some variables have embedded methods that operate on the object's contents.

Blocks and Indentation

Blocks start after a colon on the preceding line. For example:

```
if (1):
    print 'I am a new block'
```

The block continues until a new block is created, or until the indentation for the block ends.

When indenting, use either tabs or spaces—do not mix the two because this is likely to confuse the Python interpreter and people maintaining the code.

Method Calls

The most common error with a method call to an object is to assume that the call returns information. The statement

```
sorted = list.sort()
```

sorts the object **list**, but sets the value of **sorted** to the special **None**.