BONUS CHAPTER 42

Using Perl

IN THIS CHAPTER

Perl (whose name comes from the *Practical Extraction and Report Language* or the *Pathologically Eclectic Rubbish Lister*, depending on whom you speak to) is a powerful scripting tool that enables you to manage files, create reports, edit text, and perform many other tasks. Perl is included with and installed in Ubuntu by default and could be considered an integral part of the distribution because Ubuntu depends on Perl for many types of software services, logging activities, and software tools.

Perl is not the easiest of programming languages to learn because it is designed for flexibility. This chapter shows how to create and use Perl scripts on your system. You learn what a Perl program looks like, how the language is structured, and where you can find modules of prewritten code to help you write your own Perl scripts. This chapter also includes several examples of Perl used to perform a few common functions on a computer system.

Using Perl with Linux

Although originally designed as a data-extraction and report-generation language, Perl appeals to many Linux system administrators because they can use it to create utilities that fill a gap between the capabilities of shell scripts and compiled C programs. Another advantage of Perl over other UNIX tools is that it can process and extract data from binary files, whereas `sed` and `awk` cannot.

NOTE

In Perl, “there is more than one way to do it.” This is the unofficial motto of Perl, and it comes up so often that it is usually abbreviated as TIMTOWTDI.

You can use Perl at your shell's command line to execute one-line Perl programs, but most often the programs (usually ending in `.pl`) are run as commands. These programs generally work on any computer platform because Perl has been ported to nearly every operating system.

Perl programs are used to support a number of Ubuntu services, such as system logging. For example, if you install the `logwatch` package, the `logwatch.pl` program is run every morning at 6:25 a.m. by the `crond` (scheduling) daemon on your system. Other Ubuntu services supported by Perl include the following:

- Amanda for local and network backups
- Fax spooling with the `faxrunqd` program
- Printing supported by Perl document-filtering programs
- Hardware sensor monitoring setup using the `sensors-detect` Perl program

Perl Versions

Perl is installed in Ubuntu by default. You can download the code from www.perl.com and build the newest version from source if you want to, although a stable and quality release of Perl is already installed by default in Ubuntu and most (perhaps all) Linux and UNIX-like distributions, including macOS. Updated versions might appear in the Ubuntu repositories, but they're generally only security fixes that can be installed by updating your system. See Chapter 9, "Managing Software," to see how to quickly get a list of available updates for Ubuntu.

You can determine what version of Perl you have installed by typing `perl -v` at a shell prompt. When you install the latest Ubuntu distribution, you should have the latest version of Perl that was available when the software for your Ubuntu release was gathered and finalized.

Note that there was a development effort underway to replace Perl 5. The new version was to be called Perl 6, but had diverged far enough away from Perl 5 that the decision was made to rename it to Raku. You can learn about Raku at www.raku.org.

A Simple Perl Program

This section introduces a very simple Perl program example to get you started using Perl. Although trivial for experienced Perl hackers, this short example is necessary for new users who want to learn more about Perl.

To introduce you to the absolute basics of Perl programming, Listing 42.1 illustrates a simple Perl program that prints a short message.

LISTING 42.1 A Simple Perl Program

```
#!/usr/bin/perl
print 'Look at all the camels!\n';
```

Type in the program shown in the listing and save it to a file called `trivial.pl`. Then make the file executable by using the `chmod` command (see the following sidebar) and run it at the command prompt.

Command-Line Error

If you get the message `bash: trivial.pl: command not found` or `bash: ./trivial.pl: Permission denied`, you have either typed the command line incorrectly or forgotten to make `trivial.pl` executable with the `chmod` command, as shown here:

```
matthew@seymour:~$ chmod +x trivial.pl
```

You can force the command to execute in the current directory as follows:

```
matthew@seymour:~$ ./trivial.pl
```

Or you can use Perl to run the program, like this:

```
matthew@seymour:~$ perl trivial.pl
```

The sample program in the listing is a two-line Perl program. When you type in the program and run it (using Perl or by making the program executable), you are creating a Perl program, a process duplicated by Linux users around the world every day.

NOTE

`#!` is often pronounced *shebang*, which is short for `sharp` (musicians' name for the `#` character), and `bang`, which is another name for the exclamation point. This notation is also used in shell scripts. See Chapter 14, "Automating Tasks and Shell Scripting," for more information about writing shell scripts.

The `#!` line is technically not part of the Perl code at all. The `#` character indicates that the rest of the screen line is a comment. The comment is a message to the shell, telling it where it should go to find the executable to run this program. The interpreter ignores the comment line. Comments are useful for documenting scripts, like this:

```
#!/usr/bin/perl
# a simple example to print a greeting
print "hello there\n";
```

The `#` character can also be used in a quoted string and used as the delimiter in a regular expression.

A block of code such as what might appear inside a loop or a branch of a conditional statement is indicated with curly braces (`{}`). For example, here is an infinite loop:

```
#!/usr/bin/perl
# a block of code to print a greeting forever
while (1) {
        print "hello there\n";
};
```

A Perl statement is terminated with a semicolon (`;`). A Perl statement can extend over several screen lines because Perl is not concerned about white space.

The second line of the simple program in Listing 42.1 prints the text enclosed in quotation marks. `\n` is the escape sequence for a newline character.

TIP

Using the `perldoc` and `man` commands is an easy way to get more information about the version of Perl installed on your system. To learn how to use the `perldoc` command, enter the following:

```
matthew@seymour:~$ perldoc
```

To get introductory information on Perl, you can use either of these commands:

```
matthew@seymour:~$ perldoc perl
```

```
matthew@seymour:~$ man perl
```

For an overview or table of contents of Perl's documentation, use the `perldoc` command, like this:

```
matthew@seymour:~$ perldoc perltoc
```

The documentation is extensive and well organized. Perl includes a number of standard Linux manual pages as brief guides to its capabilities, but perhaps the best way to learn more about Perl is to read its `perlfunc` document, which lists all the available Perl functions and their usage. You can view this document by using the `perldoc` script and typing `perldoc perlfunc` at the command line. You can also find this document online at https://perldoc.perl.org.

Perl Variables and Data Structures

Perl is a *weakly typed* language, meaning that it does not require that you declare a data type, such as a type of value (data) to be stored in a particular variable. C, for example, makes you declare that a particular variable is an integer, a character, a structure, or whatever the case may be. Perl variables are whatever type they need to be and can change type when you need them to.

Perl Variable Types

Perl has three variable types: *scalars*, *arrays*, and *hashes*. A different character is used to signify each variable type, so you can have the same name used with each type at the same time.

A scalar variable is indicated with the `$` character, as in `$penguin`. Scalars can be numbers or strings, and they can change type as needed. If you treat a number like a string, it becomes a string. If you treat a string like a number, it is translated into a number if it makes sense to do so; otherwise, it usually evaluates to `0`. For example, the string `"76trombones"` evaluates as the number `76` if used in a numeric calculation, but the string `"polar bear"` evaluates to `0`.

A Perl array is indicated with the `@` character, as in `@fish`. An *array* is a list of values referenced by index number, starting with the first element, numbered `0`, just as in C and `awk`. Each element in the array is a scalar value. Because scalar values are indicated with the `$` character, a single element in an array is also indicated with a `$` character.

For example, `$fish[2]` refers to the third element in the `@fish` array. This tends to throw some people off but is similar to arrays in C, in which the first array element is `0`.

A hash is indicated with the `%` character, as in `%employee`. A *hash* is a list of name/value pairs. Individual elements in a hash are referenced by name rather than by index (unlike in an array). Again, because the values are scalars, the `$` character is used for individual elements.

For example, `$employee{name}` gives you one value from the hash. Two rather useful functions for dealing with hashes are `keys` and `values`. The `keys` function returns an array that contains all the keys of the hash, and `values` returns an array of the values of the hash. Using this approach, the Perl program in Listing 42.2 displays all the values in your environment, much like typing the `bash` shell's `env` command.

LISTING 42.2 Displaying the Contents of the `env` Hash

```
#!/usr/bin/perl
foreach $key (keys %ENV)  {
  print "$key = $ENV{$key}\n";
}
```

Special Variables

Perl has a variety of special variables, which usually look like punctuation—`$_`, `$!`, and `$]`—and are all extremely useful for shorthand code. `$_` is the default variable, `$!` is the error message returned by the operating system, and `$]` is the Perl version number.

`$_` is perhaps the most useful of these. You see this variable used often in this chapter. `$_` is the Perl default variable, which is used when no argument is specified. For example, the following two statements are equivalent:

```
chomp;
```

```
chomp($_);
```

The following loops are equivalent:

```
for $cow (@cattle) {
        print "$cow says moo.\n";
}

for (@cattle)        {
        print "$_ says moo.\n";
}
```

For a complete list of the special variables, see the `perlvar` man page.

Perl Operators

Perl supports a number of operators for performing various operations. There are *comparison* operators (used to compare values, as the name implies), *compound* operators (used to combine operations or multiple comparisons), *arithmetic* operators (to perform math), and special string constants.

Comparison Operators

The comparison operators used by Perl are similar to those used by C, `awk`, and the `csh` shells, and they are used to specify and compare values (including strings). A comparison operator is most often used within an `if` statement or loop. Perl has comparison operators for numbers and strings. Table 42.1 shows the numeric comparison operators and their meanings.

Table 42.1 Numeric Comparison Operators in Perl

Operator	Meaning
`==`	Is equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`<=>`	Returns `-1` if less than, `0` if equal to, and `1` if greater than
`!=`	Not equal to
`..`	Range of `>=` first operand to `<=` second operand

Table 42.2 shows the string comparison operators and their meanings.

Table 42.2 String Comparison Operators in Perl

Operator	Meaning
`eq`	Is equal to
`lt`	Less than
`gt`	Greater than
`le`	Less than or equal to
`ge`	Greater than or equal to
`ne`	Not equal to

Operator	Meaning
cmp	Returns -1 if less than, 0 if equal to, and 1 if greater than
=~	Matched by regular expression
!~	Not matched by regular expression

Compound Operators

Perl uses compound operators, similar to those used by C or awk, which can be used to combine other operations (such as comparisons or arithmetic) into more complex forms of logic. Table 42.3 shows the compound pattern operators and their meanings.

Table 42.3 Compound Pattern Operators in Perl

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT
()	Parentheses; used to group compound statements

Arithmetic Operators

Perl supports a variety of math operations. Table 42.4 summarizes these operators.

Table 42.4 Perl Arithmetic Operators

Operator	Purpose
x**y	Raises x to the y power (same as x^y)
x%y	Calculates the remainder of x/y
x+y	Adds x to y
x-y	Subtracts y from x
x*y	Multiplies x by y
x/y	Divides x by y
-y	Negates y (switches the sign of y); also known as the unary minus
++y	Increments y by 1 and uses the value (prefix increment)
y++	Uses the value of y and then increments by 1 (postfix increment)
—y	Decrements y by 1 and uses the value (prefix decrement)
y—	Uses the value of y and then decrements by 1 (postfix decrement)
x=y	Assigns the value of y to x. Perl also supports operator-assignment operators (+=, -=, *=, /=, %=, **=, and others)

You can also use comparison operators (such as == or <) and compound pattern operators (&&, ||, and !) in arithmetic statements. They evaluate to the value 0 for false and 1 for true.

Other Operators

Perl supports a number of operators that do not fit any of the prior categories. Table 42.5 summarizes these operators.

Table 42.5 Other Perl Operators

Operator	Purpose
`~x`	Bitwise NOT (changes 0 bits to 1 and 1 bits to 0 bits)
`x & y`	Bitwise AND
`x | y`	Bitwise OR
`x ^ y`	Bitwise exclusive or (XOR)
`x << y`	Bitwise shift left (shifts `x` by `y` bits)
`x >> y`	Bitwise shift right (shifts `x` by `y` bits)
`x . y`	Concatenate `y` onto `x`
`a x b`	Repeats string `a` for `b` number of times
`x, y`	Comma operator; evaluates `x` and then `y`
`x ? y : z`	Conditional expression (If `x` is true, `y` is evaluated; otherwise, `z` is evaluated.)

Except for the comma operator and conditional expression, you can also use these operators with the assignment operator, similar to the way addition (`+`) can be combined with assignment (`=`), giving `+=`.

Special String Constants

Perl supports string constants that have special meaning or cannot be entered from the keyboard.

Table 42.6 shows most of the constants supported by Perl.

Table 42.6 Perl Special String Constants

Expression	Meaning
`\\`	The means of including a backslash
`\a`	The alert or bell character
`\b`	Backspace
`\cC`	Control character (like holding the Ctrl key down and pressing the C character)
`\e`	Escape
`\f`	Formfeed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\xNN`	Indicates that `NN` is a hexadecimal number
`\0NNN`	Indicates that `NNN` is an octal (base 8) number

Conditional Statements: `if/else` and `unless`

Perl offers two conditional statements, `if` and `unless`, which function opposite one another. `if` enables you to execute a block of code only if certain conditions are met so that you can control the flow of logic through your program. Conversely, `unless` performs the statements when certain conditions are not met.

The following sections explain and demonstrate how to use these conditional statements when writing scripts for Linux.

`if`

The syntax of the Perl `if/else` structure is as follows:

```
if (condition) {
  statement or block of code
} elsif (condition) {
  statement or block of code
} else {
  statement or block of code
}
```

`condition` is a statement that returns a true or false value.

Truth is defined in Perl in a way that might be unfamiliar to you, so be careful. Everything in Perl is true except `0` (the digit zero), `"0"` (the string containing the number `0`), `""` (the empty string), and an undefined value. Note that even the string `"00"` is a true value because it is not one of the four false cases.

A *`statement or block of code`* section is executed if the test condition returns a true value.

For example, Listing 42.3 uses the `if/else` structure and shows conditional statements using the `eq` string comparison operator.

LISTING 42.3 `if/elsif/else`

```
if  ($favorite eq "chocolate") {
      print "I like chocolate too.\n";
} elsif ($favorite eq "spinach") {
      print "Oh, I do not like spinach.\n";
} else {
     print "Your favorite food is $favorite.\n";
}
```

`unless`

`unless` works just like `if` only backward. `unless` performs a statement or block if a condition is false:

```
unless ($name eq "Rich")        {
  print "Go away, you're not allowed in here!\n";
}
```

NOTE

You can restate the preceding example in more natural language, like this:

```
print "Go away!\n" unless $name eq "Rich";
```

Looping

A *loop* repeats a program action multiple times. A simple example is a countdown timer that performs a task (waiting for one second) 300 times before telling you that your egg is done boiling.

Looping constructs (also known as *control structures*) can be used to iterate a block of code as long as certain conditions apply or while the code steps through (evaluates) a list of values, perhaps using that list as arguments.

Perl has several looping constructs.

for

The `for` construct performs a *statement* (block of code) for a set of conditions defined as follows:

```
for (start condition; end condition; increment function) {
  statement(s)
}
```

The `start` condition is set at the beginning of the loop. Each time the loop is executed, the `increment` function is performed until the end condition is achieved. This looks much like the traditional `for/next` loop. The following code is an example of a `for` loop:

```
for ($i=1; $i<=10; $i++) {
      print "$i\n"
}
```

foreach

The `foreach` construct performs a statement block for each element in a list or an array:

```
@names = ("alpha","bravo","Charlie");
foreach $name (@names) {
  print "$name sounding off!\n";
}
```

The loop variable (`$name` in the example) is not merely set to the value of the array elements; it is aliased to that element. This means if you modify the loop variable, you're actually modifying the array. If no loop array is specified, the Perl default variable `$_` may be used, as shown here:

```
@names = ("alpha","bravo","Charlie");
foreach (@names) {
  print "$_ sounding off!\n";

}
```

This syntax can be very convenient, but it can also lead to unreadable code. Give a thought to the poor person who'll be maintaining your code. (It will probably be you.)

NOTE

`foreach` is frequently abbreviated as `for`.

while

`while` performs a block of statements as long as a particular condition is true, as shown in this example:

```
while ($x<10) {
     print "$x\n";
     $x++;
}
```

Remember that the condition can be anything that returns a true or false value. For example, it could be a function call, like this:

```
while ( InvalidPassword($user, $password) )        {
        print "You've entered an invalid password. Please try again.\n";
        $password = GetPassword;
}
```

until

`until` is exactly the opposite of the `while` statement. It performs a block of statements as long as a particular condition is false (or, rather, until it becomes true). The following is an example:

```
until (ValidPassword($user, $password))  {
  print "YSdpgm_m
Sdpgm_m
You have entered an invalid password. Please try again.\n";
Sdpgm_m
  $password = GetPassword;
}
```

`last` and `next`

You can force Perl to end a loop early by using a `last` statement. `last` is similar to the C `break` command; the loop is exited. If you decide you need to skip the remaining contents of a loop without ending the loop, you can use `next`, which is similar to the C `continue` command. Unfortunately, these statements do not work with `do ... while`. However, you can use `redo` to jump to a loop (marked by a label) or inside the loop where called:

```
$a = 100;
while (1) {
  print "start\n";
  TEST: {
    if (($a = $a / 2) > 2) {
      print "$a\n";
      if (—$a < 2) {
        exit;
      }
      redo TEST;
    }
  }
}
```

In this simple example, the variable `$a` is repeatedly manipulated and tested in a loop. The word `start` will be printed only once.

`do ... while` and `do ... until`

The `while` and `until` loops evaluate the conditional first. The behavior is changed by applying a `do` block before the conditional. With the `do` block, the condition is evaluated last, which results in the contents of the block always executing at least once (even if the condition is false). This is similar to the C language `do ... while (conditional)` statement.

Regular Expressions

Perl's greatest strength lies in text and file manipulation, which is accomplished by using the *regular expression* (*regex*) library. Regexes, which are quite different from the wildcard-handling and filename-expansion capabilities of the shell (see Chapter 14, "Automating Tasks and Shell Scripting"), allow complicated pattern matching and replacement to be done efficiently and easily.

For example, the following line of code replaces every occurrence of the string `bob` or the string `mary` with `fred` in a line of text:

```
$string =~ s/bob|mary/fred/gi;
```

Without going into too many of the details, Table 42.7 explains what the preceding line says.

Table 42.7 Explanation of `$string =~ s/bob|mary/fred/gi;`

Element	Explanation
`$string =~`	Performs this pattern match on the text found in the variable called `$string`.
`s`	Performs a substitution.
`/`	Begins the text to be matched.
`bob|mary`	Matches the text `bob` or `mary`. You should remember that it is looking for the text `mary`, not the word `mary`; that is, it will also match the text `mary` in the word `maryland`.
`fred`	Replaces anything that was matched with the text `fred`.
`/`	Ends replace text.
`g`	Does this substitution globally; that is, replaces the match text wherever in the string you match it (and any number of times).
`i`	The search text is not case sensitive. It matches `bob`, `Bob`, or `bOB`.
`;`	Indicates the end of the line of code.

If you are interested in the details, you can get more information using the regex (7) section of the man page by entering `man 7 regex` from the command line.

Although replacing one string with another might seem a rather trivial task, the code required to do the same thing in another language (for example, C) is rather daunting unless supported by additional subroutines from external libraries.

Access to the Shell

Perl can perform for you any process you might ordinarily perform by typing commands to the shell through the \\ syntax. For example, the code in Listing 42.4 prints a directory listing.

LISTING 42.4 Using Backticks to Access the Shell

```
$curr_dir = `pwd`;
@listing = `ls -al`;
print "Listing for $curr_dir\n";
foreach $file (@listing) {
     print "$file";
}
```

NOTE

The \\ notation uses the backtick found above the Tab key (on most keyboards), not the single quotation mark.

You can also use the `Shell` module to access the shell. `Shell` is one of the standard modules that come with Perl; it allows creation and use of a shell-like command line. The following code provides an example:

```
use Shell qw(cp);
cp ("/home/httpd/logs/access.log", :/tmp/httpd.log");
```

This code almost looks like it is importing the command-line functions directly into Perl. Although that is not really happening, you can pretend that the code is similar to a command line and use this approach in your Perl programs.

A third method of accessing the shell is via the `system` function call:

```
$rc = 0xffff & system(`cp /home/httpd/logs/access.log /tmp/httpd.log`);
if ($rc == 0) {
  print "system cp succeeded \n";
} else {
  print "system cp failed $rc\n";
}
```

The call can also be used with the `or die` clause:

```
system(`cp /home/httpd/logs/access.log /tmp/httpd.log`) == 0
  or die "system cp failed: $?"
```

However, you cannot capture the output of a command executed through the `system` function.

Modules and CPAN

A great strength of the Perl community (and the Linux community) is the fact that it is an open source community. This community support is expressed for Perl via the *Comprehensive Perl Archive Network* (*CPAN*), which is a network of mirrors of a repository of Perl code.

Most of CPAN is made up of *modules*, which are reusable chunks of code that do useful things, similar to software libraries containing functions for C programmers. These modules help speed development when building Perl programs and free Perl hackers from repeatedly reinventing the wheel when building bicycles.

Perl comes with a set of standard modules installed. Those modules should contain much of the functionality that you initially need with Perl. If you need to use a module not installed with Ubuntu, use the CPAN module (which is one of the standard modules) to download and install other modules onto your system. At https://cpan.perl.org, you can find the CPAN Multiplex Dispatcher, which attempts to direct you to the CPAN site closest to you.

Typing the following command puts you into an interactive shell that gives you access to CPAN. You can type `help` at the prompt to get more information on how to use the CPAN program:

```
matthew@seymour:~$ perl -MCPAN -e shell
```

After installing a module from CPAN (or writing one of your own), you can load that module into memory, where you can use it with the `use` function:

```
use Time::CTime;
```

`use` looks in the directories listed in the variable `@INC` for the module. In this example, `use` looks for a directory called `Time`, which contains a file called `CTime.pm`, which in turn is assumed to contain a package called `Time::CTime`. The distribution of each module should contain documentation on using that module.

For a list of all the standard Perl modules (those that come with Perl when you install it), see `perlmodlib` in the Perl documentation. You can read this document by typing `perldoc perlmodlib` at the command prompt.

Code Examples

The following sections contain a few examples of things you might want to do with Perl.

Sending Mail

You can get Perl to send email in several ways. One method that you see frequently is opening a pipe to the `sendmail` command and sending data to it (as shown in Listing 42.5). Another method is using the `Mail::Sendmail` module (available through CPAN), which uses socket connections directly to send mail (as shown in Listing 42.6). The latter method is faster because it does not have to launch an external process. Note that `sendmail` must be installed on your system for the Perl program in Listing 42.5 to work.

LISTING 42.5 Sending Mail Using `Sendmail`

```
#!/usr/bin/perl
open (MAIL, "| /usr/sbin/sendmail -t"); # Use -t to protect from users
print MAIL <<EndMail;
To: you\
From: me\
Subject: A Sample Email\nSending email from Perl is easy!\n
.
EndMail
close MAIL;
```

NOTE

The `@` sign in the email addresses must be escaped so that Perl does not try to evaluate an array of that name. That is, `dpitts@mk.net` will cause a problem, so you need to use this:

```
dpitts\<indexterm startref="iddle2799" class="endofrange"
significance="normal"/>:}]
```

The syntax used to print the mail message is called a *here document*. The syntax is as follows:

```
print <<EndText;
.....
EndText
```

The `EndText` value must be identical at the beginning and at the end of the block, including any white space.

LISTING 42.6 Sending Mail Using the `Mail::Sendmail` Module

```
#!/usr/bin/perl
use Mail::Sendmail;

%mail = ( To => "you@there.com",
  From => "me@here.com",
  Subject => "A Sample Email",
  Message => "This is a very short message"
  );

sendmail(%mail) or die $Mail::Sendmail::error;

print "OK. Log says:\n", $Mail::Sendmail::log;

use Mail::Sendmail;
```

Perl ignores the comma after the last element in the hash. It is convenient to leave it there; then, if you want to add items to the hash, you do not need to add the comma. This is purely a style decision.

Using Perl to Install a CPAN Module

You can use Perl to interactively download and install a Perl module from the CPAN archives by using the `-M` and `-e` commands. Start the process by using a Perl command like this:

```
# perl -MCPAN -e shell
```

When you press Enter, you see some introductory information, and you are asked to choose an initial automatic or manual configuration, which is required before any download or install takes place. Type `no` and press Enter to have Perl automatically configure for the download and install process; or if you want, just press Enter to manually configure for downloading and installation. If you use manual configuration, you must answer a series of questions regarding paths, caching, terminal settings, program locations, and so on. Settings are saved in a directory named `.cpan` in the current directory.

When finished, you see the CPAN prompt:

```
cpan>
```

To have Perl examine your system and then download and install a large number of modules, use the `install` keyword, specify `Bundle` at the prompt, and then press Enter, like this:

```
cpan> install Bundle::CPAN
```

To download a desired module (using the example in Listing 42.6), use the `get` keyword, like this:

```
cpan> get Mail::Sendmail
```

The source for the module is downloaded into the `.cpan` directory. You can then build and install the module by using the `install` keyword, like this:

```
cpan> install Mail::Sendmail
```

The entire process of retrieving, building, and installing a module can also be accomplished at the command line by using Perl's `-e` option, like this:

```
# perl -MCPAN   -e "install Mail::Sendmail"
```

Note also that the `@` sign in Listing 42.6 does not need to be escaped within single quotation marks (`''`). Perl does not *interpolate* (evaluate variables) within single quotation marks but does within double quotation marks and `here` strings (similar to `<<` shell operations).

Purging Logs

Many programs maintain some variety of logs. Often, much of the information in the logs is redundant or just useless. The program shown in Listing 42.7 removes all lines from a file that contain a particular word or phrase, so lines that you know are not important can be purged. For example, you might want to remove all the lines in the Apache error log that originate with your test client machine because you know those error messages were produced during testing.

LISTING 42.7 Purging Log Files

```
#!/usr/bin/perl
# Be careful using this program!
# This will remove all lines that contain a given word
# Usage:  remove <word> <file>
$word=@ARGV[0];
$file=@ARGV[1];
if ($file)  {
  # Open file for reading
  open (FILE, "$file") or die "Could not open file: $!";
      @lines=<FILE>;
  close FILE;
  # Open file for writing
  open (FILE, ">$file") or die "Could not open file for writing: $!";
  for (@lines)  {
    print FILE unless /$word/;
  } # End for
  close FILE;
} else {
  print "Usage:  remove <word> <file>\n";
}  #  End if...else
```

The code in Listing 42.7 includes a few idiomatic Perl expressions to keep it brief. It reads the file into an array by using the `<FILE>` notation; it then writes the lines back out to the file unless they match the pattern given on the command line.

The `die` function kills program operation and displays an error message if the `open` statements fail. `$!` in the error message, as mentioned earlier in this chapter, is the error message returned by the operating system. It is likely to be something like `'file not found'` or `'permission denied'`.

Posting to Usenet

If some portion of your job requires periodic postings to Usenet—an FAQ listing, for example—the following Perl program can automate the process for you. In Listing 42.8, the posted text is read in from a text file, but your input can come from anywhere.

The program shown in Listing 42.8 uses the `Net::NNTP` module, which is a standard part of the Perl distribution. You can find more documentation on the `Net::NNTP` module by entering `'perldoc Net::NNTP'` at the command line.

LISTING 42.8 Posting an Article to Usenet

```
#!/usr/bin/perl
# load the post data into @post
open (POST, "post.file");
@post = <POST>;
close POST;
# import the NNTP module
use Net::NNTP;
$NNTPhost = 'news';
# attempt to connect to the remote host;
# print an error message on failure
$nntp = Net::NNTP->new($NNTPhost)
  or die "Cannot contact $NNTPhost: $!";
# $nntp->debug(1);
$nntp->post()
  or die "Could not post article: $!";
# send the header of the post
$nntp->datasend("Newsgroups: news.announce\n");
$nntp->datasend("Subject: FAQ - Frequently Asked Questions\n");
```

```
$nntp->datasend("From: ADMIN <root>\n"
$nntp->datasend("\n\n");
# for each line in the @post array, send it
for (@post)     {
  $nntp->datasend($_);
} #  End for
$nntp->quit;
```

One-Liners

Perl excels at the one-liner. Folks go to great lengths to reduce tasks to one line of Perl code. Perl has the rather undeserved reputation of being unreadable. The fact is that you can write unreadable code in any language. Perl allows for more than one way to do something, and this leads rather naturally to people trying to find the most arcane way to do things.

Named for Randal Schwartz, a *Schwartzian* transform is a way of sorting an array by something that is not obvious. The `sort` function sorts arrays alphabetically; that is pretty obvious. What if you want to sort an array of strings alphabetically by the third word? Perhaps you want something more useful, such as sorting a list of files by file size? A Schwartzian transform creates a new list that contains the information you want to sort by, referencing the first list. You then sort the new list and use it to figure out the order that the first list should be in. Here's a simple example that sorts a list of strings by length:

```
@sorted_by_length =
  map { $_ => [0] }            # Extract original list
  sort { $a=>[1] <=> $b=>[1] } # Sort by the transformed value
  map { [$_, length($_)] }     # Map to a list of element lengths
  @list;
```

Because each operator acts on the thing immediately to the right of it, it helps to read this from right to left (or bottom to top, the way it is written here).

The first thing that acts on the list is the `map` operator. It transforms the list into a hash in which the keys are the list elements and the values are the lengths of each element. This is where you put in your code that does the transformation by which you want to sort.

The next operator is the `sort` function, which sorts the list by the values.

Finally, the hash is transformed back into an array by extracting its keys. The array is now in the desired order.

Command-Line Processing

Perl is great at parsing the output of various programs. This is a task for which many people use tools such as `awk` and `sed`. Perl gives you a larger vocabulary for performing these tasks. The following example is very simple but illustrates how you might use Perl to chop up some output and do something with it. In this example, Perl is used to list only files that are larger than 10KB:

```
matthew@seymour:~$ ls -la | perl -nae 'print "$F[8] is $F[4]\n" if $F[4] >
10000;'
```

The `-n` switch indicates that the Perl code should run for each line of the output. The `-a` switch automatically splits the output into the `@F` array. The `-e` switch indicates that the Perl code is going to follow on the command line.

Related Ubuntu and Linux Commands

You use these commands and tools often when using Perl with Linux:

- **`a2p`**—A filter used to translate `awk` scripts into Perl
- **`find2perl`**—A utility used to create Perl code from command lines using the `find` command
- **`perldoc`**—A Perl utility used to read Perl documentation
- **`s2p`**—A filter used to translate `sed` scripts into Perl
- **`vi`**—The `vi` (actually `vim`) text editor

References

- ***Learning Perl, Third Edition*** **by Randal L. Schwartz and Tom Phoenix—**The standard entry text for learning Perl.
- ***Programming Perl*, 3rd edition, by Larry Wall, Tom Christiansen, and Jon Orwant—**The standard advanced text for learning Perl.
- ***Mastering Regular Expressions*** **by Jeffrey Friedl—**Regular expressions are what make Perl so powerful.
- **www.perl.com—**This is the place to find all sorts of information about Perl, from its history and culture to helpful tips. This is also the place to download the Perl interpreter for your system.
- **https://cpan.perl.org—**CPAN is the place for you to find modules and programs in Perl. If you write something in Perl that you think is particularly useful, you can make it available to the Perl community here.
- **https://perldoc.perl.org/index-faq.html—**FAQ index of common Perl queries; this site offers a handy way to quickly search for answers about Perl.
- **https://learn.perl.org—**One of the best places to start learning Perl online.
- **http://jobs.perl.org—**If you master Perl, go to this site to look for a job.
- **www.pm.org—**The Perl Mongers are local Perl user groups. There might be one in your area.
- **www.perl.org—**This is the Perl advocacy site.

BONUS CHAPTER 43

Using Python

IN THIS CHAPTER

As PHP has come to dominate the world of web scripting, Python is increasingly dominating the domain of command-line scripting. Python's precise and clean syntax makes it one of the easiest languages to learn, and it allows programmers to code more quickly and spend less time maintaining their code. Whereas PHP is fundamentally similar to Java and Perl, Python is closer to C and Modula-3, and so it might look unfamiliar at first.

This chapter constitutes a quick-start tutorial to Python, designed to give you all the information you need to put together basic scripts and to point you toward resources that can take you further.

Two versions of Python are out in the wild right now. Python version 2.x reached end-of-life in 2019, although you are likely to see it in the wild for a while longer. Python 3.x is now the default version everywhere, including in Ubuntu, although Python 2.x will likely not yet be completely removed for Ubuntu 20.04; it will have to be installed intentionally alongside Python 3.x if it is needed.

Python 3.x is not backward compatible, and most programs written in or for 2.x need some work to run on 3.x, which is why the previous version is still available. However, most companies are working hard to update their code, such as LinkedIn, which announced in January 2020 that it "retired Python 2 and improved developer happiness" (https://engineering.linkedin.com/blog/2020/how-we-retired-python-2-and-improved-developer-happiness).

This chapter tries to note places where significant differences exist between 2.x and 3.x. If you are learning Python for the first time, start with a look at the 3.x series because it is the future.

NOTE

This chapter provides a very elementary introduction to Python. For that reason, nearly everything we cover is identical in both 2.x and 3.x. We mention the differences earlier and in a couple of notes that follow so that readers who want to learn more will not be surprised later. Unless noted otherwise, you should find what we say in this chapter applies to either version.

Python on Linux

Most versions of Linux and UNIX, including macOS, come with Python preinstalled. This is partially for convenience because it is such a popular scripting language—and it saves having to install it later if the user wants to run a script—and partially because many vital or useful programs included in Linux distributions are written in Python. For example, the Ubuntu Software Center is a Python program.

The Python binary is installed in `/usr/bin/python` (or `/usr/bin/python3`); if you run that, you enter the Python interactive interpreter, where you can type commands and have them executed immediately. Although PHP also has an interactive mode (use `php -a` to activate it), it is neither as powerful nor as flexible as Python's.

As with Perl, PHP, and other scripting languages, you can also execute Python scripts by adding a shebang line (`#!`) to the start of your scripts that point to `/usr/bin/python` and then setting the file to be executable.

The third way to run Python scripts is through `mod_python`, which is an Apache module that embeds the Python interpreter into the HTTP server, allowing you to use Python to write web applications. You can install this from the Ubuntu repositories.

We use the interactive Python interpreter for this chapter because it provides immediate feedback on commands as you type them, so it is essential that you become comfortable using it. To get started, open a terminal and run the command `python`. You should see something like this, perhaps with different version numbers and dates:

```
matthew@seymour:~$ python
Python 3.6.4 (default, Dec27 2017, 13:02:49)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` is where you type your input, and you can set and get a variable like this:

```
>>> python = 'great'
>>> python'great'
great
>>>
```

On the first line in this example, the variable `python` is set to the text `great`, and on the second line, you read back that value from the variable simply by typing the name of the variable you want to read. The third line shows Python printing the variable; on the fourth line, you are back at the prompt to type more commands. Python remembers all the variables you use while in the interactive interpreter, which means you can set a variable to be the value of another variable.

When you are done, press Ctrl+D to exit. At this point, all your variables and commands are forgotten by the interpreter, which is why complex Python programs are always saved in scripts.

The Basics of Python

Python is a language wholly unlike most others, and yet it is so logical that most people can pick it up quickly. You have already seen how easily you can assign strings, but in Python, nearly everything is that easy—as long as you remember the syntax.

Numbers

The way Python handles numbers is more precise than in some other languages. It has all the normal operators—such as `+` for addition, `-` for subtraction, `/` for division, and `*` for multiplication—but it adds `%` for modulus (division remainder), `**` for raise to the power, and `//` for floor division. It is also specific about which type of number is being used, as this example shows:

```
>>> a = 5
>>> b = 10
>>> a * b
50
>>> a / b
0
>>> b = 10.0
>>> a / b
0.5
>>> a // b
0.0
```

The first division returns `0` because both `a` and `b` are integers (whole numbers), so Python calculates the division as an integer, giving `0`. By converting `b` to `10.0`, Python considers it to be a floating-point number, and so the division is now calculated as a floating-point value, giving `0.5`. Even with `b` being a floating point, using `//`—floor division—rounds it down.

Using `**`, you can easily see how Python works with integers:

```
>>> 2 ** 30
1073741824
>>> 2 ** 31
2147483648L
```

The first statement raises `2` to the power of `30` (that is, 2 times 2 times 2 times 2…), and the second raises `2` to the power of `31`. Notice how the second number has a capital `L` on the end of it; this is Python telling you that it is a long integer. The difference between long integers and normal integers is slight but important: Normal integers can be calculated using simple instructions on the CPU, whereas long integers—because they can be as big as you need them to be—need to be calculated in software and therefore are slower.

When specifying big numbers, you need not put the `L` at the end as Python figures it out for you. Furthermore, if a number starts off as a normal number and then exceeds its boundaries, Python automatically converts it to a long integer. The same is not true the other way around: If you have a long integer and then divide it by another number so that it could be stored as a normal integer, it remains a long integer:

```
>>> num = 999999999999999999999999999999999L
>>> num = num / 1000000000000000000000000000000
>>> num
999L
```

You can convert between number types by using typecasting, like this:

```
>>> num = 10
>>> int(num)
10
>>> float(num)
10.0
>>> long(num)
10L
>>> floatnum = 10.0
>>> int(floatnum)
10
>>> float(floatnum)
10.0
>>> long(floatnum)
10L
```

You need not worry about whether you are using integers or long integers; Python handles it all for you, so you can concentrate on getting the code right. In the 3.x series, the Python integer type automatically provides extra precision for large numbers, whereas in 2.x you use a separate long integer type for numbers larger than the normal integer type is designed to handle.

More on Strings

Python stores a string as an immutable sequence of characters—a jargon-filled way of saying that "it is a collection of characters that, once set, cannot be changed without creating a new string." Sequences are important in Python. There are three primary types, of which strings are one, and they share some properties. (Mutability makes a lot of sense when you learn about lists in the next section.)

As you saw in the previous example, you can assign a value to strings in Python with just an equal sign, like this:

```
>>> mystring = 'hello';
>>> myotherstring = "goodbye";
>>> mystring'hello'
>>> myotherstring;'goodbye'
>>> test = "Are you really Jayne Cobb?"
>>> test
"Are you really Jayne Cobb?"
```

The first example encapsulates the string in single quotation marks, and the second and third examples encapsulate the string in double quotation marks. However, printing the first and second strings shows them both in single quotation marks because Python does not distinguish between the two types. The third example is the exception; it uses double quotation marks because the string itself contains a single quotation mark. Here, Python prints the string with double quotation marks because it knows it contains the single quotation mark.

Because the characters in a string are stored in sequence, you can index into them by specifying the character you are interested in. As in most other languages, in Python these indexes are zero based, which means you need to ask for character `0` to get the first letter in a string. Here is an example:

```
>>> string = "This is a test string"
>>> string
'This is a test string'
>>> string[0]
'T'
>>> string [0], string[3], string [20]
('T', 's', 'g')
```

The last line shows how, with commas, you can ask for several indexes at the same time. You could print the entire first word using the following:

```
>>> string[0], string[1], string[2], string[3]
('T', 'h', 'I', 's')
```

However, for that purpose, you can use a different concept: slicing. A *slice* of a sequence draws a selection of indexes. For example, you can pull out the first word like this:

```
>>> string[0:4]
'This'
```

The syntax here means "take everything from position 0 (including 0) and end at position 4 (excluding it)." So, `[0:4]` copies the items at indexes `0`, `1`, `2`, and `3`. You can omit either side of the indexes, and it will copy either from the start or to the end:

```
>>> string [:4]
```

```
'This'
>>> string [5:]
'is a test string'
>>> string [11:]
'est string'
```

You can also omit both numbers, and it gives you the entire sequence:

```
>>> string [:]'This is a test string'
```

Later you learn precisely why you would want to do this, but for now we look at a number of other string intrinsics that will make your life easier. For example, you can use the `+` and `*` operators to concatenate (join) and repeat strings, like this:

```
>>> mystring = "Python"
>>> mystring * 4
'PythonPythonPythonPython'
>>> mystring = mystring + " rocks! "
>>> mystring * 2
'Python rocks! Python rocks! '
```

In addition to working with operators, Python strings come with a selection of built-in methods. You can change the case of the letters with `capitalize()` (uppercases the first letter and lowercases the rest), `lower()` (lowercases them all), `title()` (uppercases the first letter in each word), and `upper()` (uppercases them all). You can also check whether strings match certain cases with `islower()`, `istitle()`, and `isupper()`; this also extends to `isalnum()` (which returns `true` if the string is letters and numbers only) and `isdigit()` (which returns `true` if the string is all numbers).

This example demonstrates some of these in action:

```
>>> string
'This is a test string'
>>> string.upper()
'THIS IS A TEST STRING'
>>> string.lower()
'this is a test string'
>>> string.isalnum()
False
>>> string = string.title()
>>> string
'This Is A Test String'
```

Why did `isalnum()` return `false`? Doesn't the string contain only alphanumeric characters? Well, no. There are spaces in there, which is what is causing the problem. More importantly, this example calls `upper()` and `lower()`, and they are not changing the contents of the string but just returning the new value. So, to change the string from `This` `is a test string` to `This Is A Test String`, you have to assign it back to the string variable.

Another really useful and kind of cool thing you can do with strings is triple quoting. This way, you can easily create strings that include both double- and single-quoted strings, as well as strings that span more than one line, without having to escape a bunch of special characters. Here's an example:

```
>>>foo = """
..."foo"
...'bar'
...baz
...blippy
..."""
>>>foo
'\n"foo"\n\'bar\'\nbaz\nblippy\n'
>>>
```

Although this is intended as an introductory guide, the capability to use negative indexes with strings and slices is a pretty neat feature that deserves a mention. Here's an example:

```
>>>bar="news.google.com"
>>>bar[-1]
'm'
>>>bar[-4:]
'.com'
>>>bar[-10:-4]
'google'
>>>
```

One difference between Python 2.x and 3.x is the introduction of new string types and method calls. This doesn't affect what we share here but is something to pay attention to as you work with Python.

Lists

Python's built-in list data type is a sequence, like strings. However, lists are mutable, which means you can change them. A list is like an array in that it holds a selection of elements in a given order. You can cycle through them, index into them, and slice them:

```
>>> mylist = ["python", "perl", "php"]
>>> mylist
['python', 'perl', 'php']
>>> mylist + ["java"]
["python", 'perl', 'php', 'java']
>>> mylist * 2
['python', 'perl', 'php', 'python', 'perl', 'php']
>>> mylist[1]
```

```
'perl'
>>> mylist[1] = "c++"
>>> mylist[1]
'c++'
>>> mylist[1:3]
['c++', 'php']
```

The brackets notation is important: You cannot use parentheses (`()`) or braces (`{}`) for lists. Using `+` for lists is different from using `+` for numbers. Python detects that you are working with a list and appends one list to another. This is known as *operator overloading*, and it is part of what makes Python so flexible.

Lists can be *nested*, which means you can put a list inside a list. However, this is where mutability starts to matter, and so this might sound complicated. Recall that an immutable string sequence is a collection of characters that, once set, cannot be changed without creating a new string. Lists are mutable, as opposed to immutable, which means you can change a list without creating a new list. This becomes important because Python, by default, copies only a reference to a variable rather than the full variable. Consider this example:

```
>>> list1 = [1, 2, 3]
>>> list2 = [4, list1, 6]
>>> list1
[1, 2, 3]
>>> list2
[4, [1, 2, 3], 6]
```

This example shows a nested list. `list2` contains `4`, then `list1`, then `6`. When you print the value of `list2`, you can see that it also contains `list1`. Now, let's proceed on from that:

```
>>> list1[1] = "Flake"
>>> list2
[4, [1, 'Flake', 3], 6]
```

In the first line, you set the second element in `list1` (remember that sequences are zero based!) to be `Flake` rather than `2`; then you print the contents of `list2`. As you can see, when `list1` changes, `list2` is updated, too. The reason for this is that `list2` stores a reference to `list1` as opposed to a copy of `list1`; they share the same value.

You can see that this works both ways by indexing twice into `list2`, like this:

```
>>> list2[1][1] = "Caramello"
>>> list1
[1, 'Caramello', 3]
```

The first line says, "Get the second element in `list2 (list1)` and the second element of that list and set it to be 'Caramello'." Then `list1`'s value is printed, and you can see that it has changed. This is the essence of mutability: You are changing a list without creating a new list. However, editing a string creates a new string, leaving the old one unaltered. Here is an example:

```
>>> mystring = "hello"
>>> list3 = [1, mystring, 3]
>>> list3
[1, 'hello', 3]
>>> mystring = "world"
>>> list3
[1, 'hello', 3]
```

Of course, this raises the question of how you copy without references when references are the default. The answer, for lists, is that you use the `[:]` slice, covered earlier in this chapter. This slices from the first element to the last, inclusive, essentially copying it without references. Here is how it looks:

```
>>> list4 = ["a", "b", "c"]
>>> list5 = list4[:]
>>> list4 = list4 + ["d"]
>>> list5
['a', 'b', 'c']
>>> list4
['a', 'b', 'c', 'd']
```

Lists have their own collections of built-in methods, such as `sort()`, `append()`, and `pop()`. The latter two add and remove single elements from the end of the list, and `pop()` also returns the removed element. Here is an example:

```
>>> list5 = ["nick", "paul", "Julian", "graham"]
>>> list5.sort()
>>> list5
['graham', 'julian', 'nick', 'paul'']
>>> list5.pop()'paul'
>>> list5
['graham', 'julian', 'nick']

>>> list5.append("Rebecca")
```

In addition, one interesting method of strings returns a list: `split()`. This takes a character to split by and then gives you a list in which each element is a chunk from the string. Here is an example:

```
>>> string = "This is a test string";
>>> string.split(" ")
['This', 'is', 'a', 'test', 'string']
```

Lists are used extensively in Python, although this is slowly changing as the language matures. The way lists are compared and sorted has changed in 3.x, so be sure to check the latest documentation when you attempt to perform these tasks.

Dictionaries

Unlike a list, a dictionary is a collection with no fixed order. Instead, it has a *key* (the name of the element) and a *value* (the content of the element), and Python places a dictionary wherever it needs to for maximum performance. When defining dictionaries, you need to use braces ({}) and colons (:). You start with an opening brace and then give each element a key and a value, separated by a colon, like this:

```
>>> mydict = { "perl" : "a language", "php" : "another language" }
>>> mydict
{'php': 'another language', 'perl': 'a language'}
```

This example has two elements, with keys `perl` and `php`. However, when the dictionary is printed, you find that `php` comes before `perl`; Python hasn't respected the order in which you entered them. You can index into a dictionary by using the normal code:

```
>>> mydict["perl"]'a language'
```

However, because a dictionary has no fixed sequence, you cannot take a slice or an index by position.

Like lists, dictionaries are mutable and can also be nested; however, unlike with lists, you cannot merge two dictionaries by using `+`. This is because dictionary elements are located using the key. Therefore, having two elements with the same key would cause a clash. Instead, you should use the `update()` method, which merges two arrays by overwriting clashing keys.

You can also use the `keys()` method to return a list of all the keys in a dictionary.

Subtle differences exists between how dictionaries are used in 3.x compared to in 2.x, such as the need to make list calls to produce all values at once so they may be printed. Again, be sure to check the latest documentation as you move ahead because there are several of these changes in functionality, with some tools now behaving differently or even disappearing and other new dictionary tools being added. Some of those features have been backported; for example, Python 2.7 received support for ordered dictionaries, backported from Python 3.1 (see https://docs.python.org/dev/whatsnew/2.7.html#pep-0372).

Conditionals and Looping

So far, we have just been looking at data types, which should show you how powerful Python's data types are. However, you cannot write complex programs without conditional statements and loops.

Python has most of the standard conditional checks, such as `>` (greater than), `<=` (less than or equal to), and `==` (equal), but it also adds some new ones, such as `in`. For example, you can use `in` to check whether a string or a list contains a given character or element:

```
>>> mystring = "J Random Hacker"
>>> "r" in mystring
True
>>> "Hacker" in mystring
True
>>> "hacker" in mystring
False
```

This example demonstrates how strings are case sensitive. You can use the equal operator for lists, too:

```
>>> mylist = ["soldier", "sailor", "tinker", "spy"]
>>> "tailor" in mylist
False
```

Other comparisons on these complex data types are done item by item:

```
>>> list1 = ["alpha", "beta", "gamma"]
>>> list2 = ["alpha", "beta", "delta"]
>>> list1 > list2
True
```

In this case, `list1`'s first element (`alpha`) is compared against `list2`'s first element (`alpha`) and, because they are equal, the next element is checked. That is equal also, so the third element is checked, which is different. The `g` in `gamma` comes after the `d` in `delta` in the alphabet, so `gamma` is considered greater than `delta`, and `list1` is considered greater than `list2`.

Loops come in two types, and both are equally flexible. For example, the `for` loop can iterate through letters in a string or elements in a list:

```
>>> string = "Hello, Python!"
>>> for s in string: print s,
...
H e l l o ,   P y t h o n !
```

The `for` loop takes each letter in a string and assigns it to `s`. This is then printed to the screen using the `print` command, but note the comma at the end; it tells Python not to insert a line break after each letter. The ellipsis (`...`) is there because Python allows you to enter more code in the loop; you need to press Enter again here to have the loop execute.

You can use the same construct for lists:

```
>>> mylist = ["andi", "rasmus", "zeev"]
>>> for p in mylist: print p
```

```
...
andi
rasmus
zeev
```

Without the comma after the `print` statement, each item is printed on its own line.

The other loop type is the `while` loop, and it looks similar:

```
>> while 1: print "This will loop forever!"
...
This will loop forever!
This will loop forever!
This will loop forever!
This will loop forever!
This will loop forever!
(etc)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyboardInterrupt
>>>
```

That is an infinite loop (it carries on printing that text forever), so you need to press Ctrl+C to interrupt it and regain control.

If you want to use multiline loops, you need to get ready to use your Tab key: Python handles loop blocks by recording the level of indent used. Some people find this odious; others admire it for forcing clean coding on users. Most of us, though, just get on with programming.

Here is an example:

```
>>> i = 0
>>> while i < 3:
...     j = 0
...     while j < 3:
...             print "Pos: " + str(i) + "," + str(j) + ")"
...             j += 1
...     i += 1
...
Pos: (0,0)
Pos: (0,1)
Pos: (0,2)
Pos: (1,0)
Pos: (1,1)
Pos: (1,2)
Pos: (2,0)
Pos: (2,1)
Pos: (2,2)
```

You can control loops by using the break and continue keywords. break exits the loop and continues processing immediately afterward, and continue jumps to the next loop iteration.

Again, many subtle changes have occurred from 2.x to 3.x. Most conditionals and looping work the same, but because we mention it in this section, we should note that print is no longer a statement in 3.x and is now a function call. That provides a nice segue to our discussion of functions.

Functions

Some languages—such as PHP—read and process an entire file before executing it, which means you can call a function before it is defined because the compiler reads the definition of the function before it tries to call it. Python is different: If the function definition has not been reached by the time you try to call it, you get an error. The reason behind this behavior is that Python actually creates an object for your function, and that in turn means two things. First, you can define the same function several times in a script and have the script pick the right one at runtime. Second, you can assign the function to another name just by using =.

A function definition starts with def, then the function name, then parentheses and a list of parameters, and then a colon. The contents of a function need to be indented at least one level beyond the definition. So, using function assignment and dynamic declaration, you can write a script that prints the right greeting in a roundabout manner:

```
>>> def hello_english(Name):
...     print "Hello, " + Name + "!"
...
>>> def hello_hungarian(Name):
...     print "Szia, " + Name + "!"
...
>>> hello = hello_hungarian
>>> hello("Paul")
Szia, Paul!
>>> hello = hello_english
>>> hello("Paul")
```

Notice that function definitions include no type information. Functions are typeless, as mentioned previously. The upside of this is that you can write one function to do several things:

```
>>> def concat(First, Second):
...     return First + Second
...
>>> concat(["python"], ["perl"])
['python', 'perl']
>>> concat("Hello, ", "world!")'Hello, world!'
```

This example demonstrates how the `return` statement sends a value back to the caller and also how a function can do one thing with lists and another thing with strings. The magic here is being accomplished by the objects. You write a function that tells two objects to add themselves together, and the objects intrinsically know how to do that. If they do not—if, perhaps, the user passes in a string and an integer—Python catches the error for you. However, it is this hands-off, "let the objects sort themselves out" attitude that makes functions so flexible. The `concat()` function could conceivably concatenate strings, lists, or *zonks*—a data type someone created herself that allows adding. The point is that Python does not limit what your function can do. As clichéd as it might sound, the only limit is your imagination.

Object Orientation

After having read this far, you should not be surprised to hear that Python's object orientation is flexible and likely to surprise you if you have been using C-like languages for some time.

The best way to learn Python *object-oriented programming (OOP)* is to just do it. Here is a basic script that defines a class, creates an object of that class, and calls a function:

```
class Dog(object):
    def bark(self):
        print "Woof!"

fluffy = Dog()
fluffy.bark()
```

Defining a class starts, predictably, with the `class` keyword followed by the name of the class you are defining and a colon. The contents of that class need to be indented one level so that Python knows where it stops. Note that the object inside parentheses is there for object inheritance, which is discussed later. For now, the least you need to know is that if your new class is not based on an existing class, you should put `object` inside parentheses, as shown in the previous code.

Functions inside classes work in much the same way as normal functions do (although they are usually called *methods*); the main difference is that they should all take at least one parameter, usually called `self`. This parameter is filled with the name of the object the function was called on, and you need to use it explicitly.

Creating an instance of a class is done by assignment. You do not need any new keyword, as in some other languages; providing the parentheses makes the creation known. Calling a function of that object is done using a period and the name of the class to call, with any parameters being passed inside parentheses.

Class and Object Variables

Each object has its own set of functions and variables, and you can manipulate those variables independently of objects of the same type. In addition, a class variable may be set to a default value for all classes, in which case it can be manipulated globally.

This script demonstrates two objects of the `dog` class being created, each with its own name:

```
class Dog(object):
     name = "Lassie"
     def bark(self):
         print self.name + " says 'Woof!'"
     def set_name(self, name):
         self.name = name
fluffy = Dog()
fluffy.bark()
poppy = Dog()
poppy.set_name("Poppy")
poppy.bark()
```

This outputs the following:

```
Lassie says 'Woof!'
Poppy says 'Woof!'
```

Here, each dog starts with the name `Lassie`, but it gets customized. Keep in mind that, by default, Python assigns by reference, meaning each object has a reference to the class's name variable, and as you assign that with the `set_name()` method, that reference is lost. This means that any references you do not change can be manipulated globally. Thus, if you change a class's variable, it also changes in all instances of that class that have not set their own value for that variable. Consider this example:

```
class Dog(object):
    name = "Lassie"
    color = "brown"
fluffy = Dog()
poppy = Dog()
print fluffy.color
Dog.color = "black"
print poppy.color
fluffy.color = "yellow"
print fluffy.color
print poppy.color
```

So, the default color of dogs is `brown`—both the `fluffy` and `poppy Dog` objects start off as `brown`. Then, using `Dog.color`, you set the default color to be `black`, and because neither of the two objects has set its own color value, they are updated to be `black`. The third-to-last line uses `poppy.color` to set a custom color value for the `poppy` object: `poppy` becomes `yellow`, whereas `fluffy` and the `Dog` class in general remain `black`.

Constructors and Destructors

To help you automate the creation and deletion of objects, you can easily override two default methods: `__init__` and `__del__`. These are the methods that Python calls when a class is being instantiated and freed, known as the *constructor* and *destructor*, respectively.

Having a custom constructor is great for when you need to accept a set of parameters for each object being created. For example, you might want each dog to have its own name on creation, and you could implement that with this code:

```
class Dog(object):
    def __init__(self, name):
        self.name = name
fluffy = Dog("Fluffy")
print fluffy.name
```

If you do not provide a name parameter when creating the `Dog` object, Python reports an error and stops. You can, of course, ask for as many constructor parameters as you want, although it is usually better to ask only for the ones you need and have other functions to fill in the rest.

On the other side of things is the destructor method, which enables you to have more control over what happens when an object is destroyed. Using a constructor and destructor, this example shows the life cycle of an object by printing messages when it is created and deleted:

```
class dog(object):
    def __init__(self, name):
        self.name = name
        print self.name + " is alive!"
    def __del__(self):
        print self.name + " is no more!"
fluffy = Dog("Fluffy")
```

The destructor is here to give you the chance to free up resources allocated to the object or perhaps log something to a file. Note that although it is possible to override the `__del__` method, doing so is very dangerous because it is not guaranteed that `__del__` methods are called for objects that still exist when the interpreter exits. The official Python documentation explains this well at https://docs.python.org/reference/datamodel.html#object.__del__.

Class Inheritance

Python allows you to reuse your code by inheriting one class from one or more others. For example, lions, tigers, and bears are all mammals and so share a number of similar properties. In that scenario, you do not want to have to copy and paste functions between them; it is smarter (and easier) to have a mammal class that defines all the shared functionality and then inherit each animal from that.

Consider the following code:

```
class Car(object):
    color = "black"
    speed = 0
    def accelerate_to(self, speed):
        self.speed = speed
    def set_color(self, color):
        self.color = color
mycar = Car()
print mycar.color
```

This example creates a `Car` class with a default color and provides a `set_color()` function so that people can change their own colors. Now, what do you drive to work? Is it a car? Sure it is, but chances are it is a Ford, or a Dodge, or a Jeep, or some other make; you do not get cars without a make. On the other hand, you do not want to have to define a class `Ford` and give it the methods `accelerate_to()`, `set_color()`, and however many other methods a basic car has and then do the same thing for Ferrari, Nissan, and so on.

The solution is to use inheritance: Define a `car` class that contains all the shared functions and variables of all the makes and then inherit from that. In Python, you do this by putting the name of the class from which to inherit inside parentheses in the `class` declaration, like this:

```
class Car(object):
    color = "black"
    speed = 0
    def accelerate_to(self, speed):
        self.speed = speed
    def set_color(self, color):
        self.color = color
class Ford(Car): pass
class Nissan(Car): pass
mycar = Ford()
print mycar.color
```

The `pass` directive is an empty statement; it means the class contains nothing new. However, because the `Ford` and `Nissan` classes inherit from the `car` class, they get `color`, `speed`, `accelerate_to()`, and `set_color()` provided by their parent class. (Note that you do not need objects after the class names for `Ford` and `Nissan` because they are inherited from an existing class: `car`.)

By default, Python gives you all the methods the parent class has, but you can override that by providing new implementations for the classes that need them. Here is an example:

```
class Modelt(car):
    def set_color(self, color):
        print "Sorry, Model Ts only come in black!"

myford Ford()
Ford.set_color("green")
mycar = Modelt()
mycar.set_color("green")
```

The first car is created as a `Ford`, so `set_color()` works fine because it uses the method from the `car` class. However, the second car is created as a `Modelt`, which has its own `set_color()` method, so the call fails.

This suggests an interesting scenario: What do you do if you have overridden a method and yet want to call the parent's method also? If, for example, changing the color of a `Model T` were allowed but just cost extra, you would want to print a message saying "You owe $50 more" but then change the color. To do this, you need to use the `class` object from which the current class is inherited (`Car` in this case). Here's an example:

```
class Modelt(Car):
    def set_color(self, color):
        print "You owe $50 more"
        Car.set_color(self, color)

mycar = Modelt()
mycar.set_color("green")
print mycar.color
```

This prints the message and then changes the color of the car.

The Standard Library and the Python Package Index

A default Python install includes many modules (blocks of code) that enable you to interact with the operating system, open and manipulate files, parse command-line options, perform data hashing and encryption, and much more. This is one of the reasons most commonly cited when people are asked why they like Python so much: It comes stocked to the gills with functionality you can take advantage of immediately. In fact, the number of modules included in the Standard Library is so high that entire books have been written about them.

For unofficial scripts and add-ons for Python, the recommended starting place is called the *Python Package Index* (*pyPI*), at https://pypi.python.org/pypi. There you can find more than 10,000 public scripts and code examples for everything from mathematics to games.

References

- **www.python.org**—The Python website is packed with information and updated regularly. This should be your first stop, if only to read the latest Python news.
- **www.jython.org**—Python also has an excellent Java-based interpreter to allow it to run on other platforms.
- **www.ironpython.com**—If you prefer Microsoft .NET, check out this site.
- **www.montypython.com—**Guido van Rossum borrowed the name for his language from *Monty Python's Flying Circus*, and as a result many Python code examples use oblique *Monty Python* references (spam and eggs are quite common, for example). A visit to the official *Monty Python* site to hone your Python knowledge is highly recommended.
- **www.zope.org—**This is the home page of the Zope *content management system* (*CMS*). One of the most popular CMSs around, Zope is written entirely in Python.

BONUS CHAPTER 44

Using PHP

IN THIS CHAPTER

This chapter introduces you to the world of PHP programming from the point of view of using it as a web scripting language and as a command-line tool. PHP originally stood for *Personal Home Page* because it was a collection of Perl scripts designed to ease the creation of guest books, message boards, and other interactive scripts commonly found on home pages. However, since those early days, it has received major updates and revisions.

Part of the success of PHP has been its powerful integration with databases; its earliest uses nearly always took advantage of a database back end.

NOTE

Many packages for PHP are available from the Ubuntu repositories. The basic package you want to install is just called `php`, and installing it brings in other packages as dependencies.

Introduction to PHP

In terms of the way it looks, PHP is a cross between Java and Perl, a successful merger of the best aspects of both into one language. The Java parts include a powerful object-orientation system, the capability to throw program exceptions, and the general style of writing that both languages borrowed from C. Borrowed from Perl is the "it should just work" mentality where ease of use is favored over strictness. As a result, you will find a lot of "there is more than one way to do it" in PHP. This also means that it is possible to accomplish tasks in ways that are less than ideal or without consideration for good security. Many criticize PHP for this, but for simple tasks or if written carefully, it can be a pretty good language and is easy to understand and use, especially for quick website creation.

Entering and Exiting PHP Mode

Unlike with PHP's predecessors, you embed your PHP code inside your HTML as opposed to the other way around. Before PHP, many websites had standard HTML pages for most of their content, linking to Perl CGI pages to do back-end processing when needed. With PHP, all your pages are capable of processing and containing HTML. This is a huge factor in PHP's popularity.

Each PHP file is processed by PHP, which looks for code to execute. PHP considers all the text it finds to be HTML until it finds one of four things:

- `<?php`
- `<?`
- `<%`
- `<script language="php">`

The first option is the preferred method of entering PHP mode because it is guaranteed to work.

When in PHP mode, you can exit it by using `?>` (for `<?php` and `<?`); `%>` (for `<%`); or `</script>` (for `<script language="php">`). This code example demonstrates entering and exiting PHP mode:

In HTML mode

```
<?php
  echo "In PHP mode";
?>
In HTML mode
In <?php echo "PHP"; ?> mode
```

Variables

Every variable in PHP starts with a dollar sign (`$`). Unlike many other languages, PHP does not have different types of variable for integers, floating-point numbers, arrays, or Booleans. They all start with a `$`, and all are interchangeable. As a result, PHP is a weakly typed language, which means you do not declare a variable as containing a specific type of data; you just use it however you want to.

Save the code in Listing 44.1 into a new file called `ubuntu1.php`.

LISTING 44.1 Testing Types in PHP

```
<?php
  $i = 10;
  $j = "10";
  $k = "Hello, world";
  echo $i + $j;
  echo $i + $k;
?>
```

To run this script, bring up a console and browse to where you saved it. Then type this command:

```
matthew@seymour:~$ php ubuntu1.php
```

If PHP is installed correctly, you should see the output `2010`, which is really two things. The `20` is the result of 10 + 10 (`$i` plus `$j`), and the `10` is the result of adding 10 to the text string `Hello, world`. Neither of those operations are really straightforward. Whereas `$i` is set to the number `10`, `$j` is actually set to be the text value `"10"`, which is not the same thing. Adding 10 to 10 gives 20, as you would imagine, but adding 10 to `"10"` (the string) forces PHP to convert `$j` to an integer on-the-fly before adding it.

Running `$i + $k` adds another string to a number, but this time the string is `Hello, world` and not just a number inside a string. PHP still tries to convert it, though, and converting any non-numerical string into a number converts it to 0. So, the second `echo` statement ends up saying `$i + 0`.

As you should have guessed by now, calling `echo` outputs values to the screen. Right now, that prints directly to your console, but internally PHP has a complex output mechanism that enables you to print to a console, send text through Apache to a web browser, send data over a network, and more.

Now that you have seen how PHP handles variables of different types, it is important that you understand the various of types available to you, as shown in Table 44.1.

Table 44.1 PHP Variable Types

Type	Stores
`integer`	Whole numbers; for example, `1`, `9`, or `324809873`
`float`	Fractional numbers; for example, `1.1`, `9.09`, or `3.141592654`
`string`	Characters; for example, `"a"`, `"sfdgh"`, or `"Ubuntu Unleashed"`
`boolean`	`true` or `false`
`array`	Several variables of any type

The first four variables in Table 44.1 can be thought of as simple variables and the last two as complex variables. Arrays are simply collections of variables. You might have an array of numbers (for example, the ages of all the children in a class); an array of strings (for example, the names of all Wimbledon tennis champions); or even an array of arrays, known as a *multidimensional array*. Arrays are covered in more depth in the next section because they are unique in the way they are defined.

An object is used to define and manipulate a set of variables that belong to a unique entity. Each object has its own personal set of variables, as well as functions that operate on those variables. Objects are commonly used to model real-world things. You might define an object that represents a TV, with variables such as `$CurrentChannel` (probably an integer), `$SupportsHiDef` (a Boolean), and so on.

Of all the complex variables, the easiest to grasp are resources. PHP has many extensions available to it that allow you to connect to databases, manipulate graphics, or even make calls to Java programs. Because they are all external systems, they need to have types of data unique to them that PHP cannot represent using any of the six other data types. So, PHP stores their custom data types in resources—data types that are meaningless to PHP but can be used by the external libraries that created them.

Arrays

Arrays are one of our favorite parts of PHP because the syntax is smart and easy to read and yet manages to be as powerful as you could want. You need to know four pieces of jargon to understand arrays:

- An array is made up of many *elements*.
- Each element has a *key* that defines its place in the array. An array can have only one element with a given key.
- Each element also has a *value*, which is the data associated with the key.
- Each array has a *cursor*, which points to the current key.

The first three are used regularly; the last one less often. The array cursor is covered later in this chapter, in the section "Basic Functions," but we look at the other three now. With PHP, your keys can be almost anything: integers, strings, objects, or other arrays. You can even mix and match the keys so that one key is an array, another is a string, and so on. The one exception to all this is floating-point numbers; you cannot use floating-point numbers as keys in your arrays.

There are two ways of adding values to an array: with the `[]` operator, which is unique to arrays; and with the `array()` pseudo-function. You should use `[]` when you want to add items to an existing array and use `array()` to create a new array.

To sum all this up in code, Listing 44.2 shows a script that creates an array without specifying keys, adds various items to it both without keys and with keys of varying types, does a bit of printing, and then clears the array.

LISTING 44.2 Manipulating Arrays

```
<?php
  $myarr = array(1, 2, 3, 4);

  $myarr[4] = "Hello";
  $myarr[] = "World!";
  $myarr["elephant"] = "Wombat";
  $myarr["foo"] = array(5, 6, 7, 8);

  echo $myarr[2];
  echo $myarr["elephant"];
  echo $myarr["foo"][1];

  $myarr = array();
?>
```

The initial array is created with four elements, to which we assign the values `1`, `2`, `3`, and `4`. Because no keys are specified, PHP automatically assigns keys for us, starting at `0` and counting upward—giving keys `0`, `1`, `2`, and `3`. Then we add a new element with the `[]` operator, specifying `4` as the key and `"Hello"` as the value. Next, `[]` is used again to add an element with the value `"World!"` and no key and then again to add an element with the key `"elephant"` and the value `"wombat"`. The line after that demonstrates using a string key with an array value—an array inside an array (a multidimensional array).

The next three lines demonstrate reading back from an array, first using a numerical key, then using a string key, and then using a string key and a numerical key. Remember that the `"foo"` element is an array in itself, so that third reading line retrieves the array and then prints the second element (arrays start at `0`, remember). The last line blanks the array by simply using `array()` with no parameters, which creates an array with elements and assigns it to `$myarr`.

The following is an alternative way of using `array()` that allows you to specify keys along with their values:

```
$myarr = array("key1" => "value1", "key2" => "value2",
7 => "foo", 15 => "bar");
```

Which method you choose really depends on whether you want specific keys or want PHP to pick them for you.

Constants

Constants are frequently used in functions that require specific values to be passed in. For example, a popular function is `extract()`, which takes all the values in an array and places them into variables in their own right. You can choose to change the name of the variables as they are extracted by using the second parameter; send it `0`, and it overwrites variables with the same names as those being extracted; send it `1`, and it skips variables with the same names; send it `5`, and it prefixes variables only if they exist already; and so on. Of course, no one wants to have to remember a lot of numbers for each function, so you can instead use `EXTR_OVERWRITE` for `0`, `EXTR_SKIP` for `1`, `EXTR_PREFIX_IF_EXISTS` for `5`, and so on, which is much easier.

You can create constants of your own by using the `define()` function. Unlike variables, constants do not start with a dollar sign. Code to define a constant looks like this:

```
<?php
  define("NUM_SQUIRRELS", 10);
  define("PLAYER_NAME", "Jim");
  define("NUM_SQUIRRELS_2", NUM_SQUIRRELS);
  echo NUM_SQUIRRELS_2;
?>
```

This script demonstrates how you can set constants to numbers, strings, or even the values of other constants (although this last option doesn't really get used much).

References

You can use the equal sign (=) to copy the value from one variable to another so that each one has a copy of the value. Another option here is to use references, which is where a variable does not have a value of its own but instead points to another variable. This enables you to share values and have variables mutually update themselves.

To copy by reference, use the & symbol, as follows:

```
<?php
  $a = 10;
  $b = &$a;
  echo $a . "\n";
  echo $b . "\n";

  $a = 20;
  echo $a . "\n";
  echo $b . "\n";

  $b = 30;
  echo $a . "\n";
  echo $b . "\n";
?>
```

If you run this script, you will see that updating `$a` also updates `$b` and also that updating `$b` updates `$a`.

Comments

Adding short comments to your code is recommended and usually a requirement in larger software houses. In PHP, you have three options for commenting style: `//`, `/* */`, and `#`. The first option (two slashes) instructs PHP to ignore everything until the end of the line. The second (a slash and an asterisk) instructs PHP to ignore everything until it reaches `*/`. The last (a hash symbol) works like `//` and is included because it is common among shell scripting languages.

This code example demonstrates the difference between `//` and `/* */`:

```
<?php
  echo "This is printed!";
  // echo "This is not printed";
  echo "This is printed!";
  /* echo "This is not printed";
  echo "This is not printed either"; */
?>
```

Using `//` is generally preferred because it is a known quantity. It is easy to introduce coding errors with `/* */` by losing track of where a comment starts and ends.

NOTE

Contrary to popular belief, having comments in your PHP script has almost no effect on the speed at which the script executes. What little speed difference exists is wholly removed if you use a code cache.

Escape Sequences

Some characters cannot be typed, and yet you will almost certainly want to use some of them from time to time. For example, you might want to use an ASCII character for a new line, but you cannot type it. Instead, you need to use an escape sequence: `\n`. Similarly, you can print a carriage return character with `\r`. It is important to know both of these because, on the Windows platform, you need to use `\r\n` to get a new line. If you do not plan to run your scripts anywhere other than your local Ubuntu or other Linux environment, you need not worry about this.

Going back to the first script you wrote, recall that it printed `2010` because you added 10 + 10 and then 10 + 0. You can rewrite that using escape sequences, like this:

```
<?php
  $i = 10;
  $j = "10";
  $k = "Hello, world";
  echo $i + $j;
  echo "\n";
  echo $i + $k;
  echo "\n";
?>
```

This time PHP prints a new line after each of the numbers, making it obvious that the output is `20` and `10` rather than `2010`. Note that the escape sequences must be used in double quotation marks because they will not work in single quotation marks.

Three common escape sequences are `\\`, which means “ignore the backslash”; `\"`, which means “ignore the double quote”; and `\'`, which means “ignore the single quote.” This is important when strings include quotation marks inside them. If we had a string such as `"Are you really Conan O'Brien?"`, which has a single quotation mark in it, this code would not work:

```
<?php
  echo 'Are you really Conan O'Brien?';
?>
```

PHP would see the opening quotation mark, read all the way up to the *O* in O’Brien, and then see the quotation mark following the *O* as being the end of the string. The Brien? part would appear to be a fragment of text and would cause an error. You have two options here: You can either surround the string in double quotation marks or escape the single quotation mark with `\'`. The escaping route looks like this:

```
echo 'Are you really Conan O\'Brien?';
```

Although escape sequences are a clean solution for small text strings, be careful not to overuse them. HTML is particularly full of quotation marks, and escaping them can get messy, as you can see in this example:

```
$mystring = "<img src=\"foo.png\" alt=\"My picture\"
width=\"100\" height=\"200\" />";
```

In such a situation, you are better off using single quotation marks to surround the text simply because it is a great deal easier on the eye.

Variable Substitution

PHP allows you to define strings using three methods: single quotation marks, double quotation marks, or heredoc notation. Heredoc is not discussed in this chapter because it is fairly rare compared to the other two methods, but single quotation marks and double quotation marks work identically, with one minor exception: variable substitution.

Consider the following code:

```
<?php
  $age = 25
  echo "You are ";
  echo $age;
?>
```

This is a particularly clumsy way to print a variable as part of a string. Fortunately, if you put a variable inside a string, PHP performs *variable substitution*, replacing the variable with its value. That means you can rewrite the code like this:

```
<?php
  $age = 25
  echo "You are $age";
?>
```

The output is the same. The difference between single quotation marks and double quotation marks is that single-quoted strings do not have their variables substituted. Here's an example:

```
<?php
  $age = 25
  echo "You are $age";
  echo 'You are $age';
?>
```

The first `echo` prints `You are 25`, and the second one prints `You are $age`.

Operators

Now that you have data values to work with, you need some operators to use, too. You have already used `+` to add variables together, but many other operators in PHP handle arithmetic, comparison, assignment, and other operators. *Operator* is just a fancy word for something that performs an operation, such as addition or subtraction. However, *operand* might be new to you. Consider this operation:

```
$a = $b + c;
```

In this operation, `=` and `+` are operators, and `$a`, `$b`, and `$c` are operands. Along with `+`, you also already know - (subtract), `*` (multiply), and `/` (divide), but Table 44.2 provides some more.

Table 44.2 PHP Operators

Operator	What It Does
`=`	Assigns the right operand to the left operand.
`==`	Returns `true` if the left operand is equal to the right operand.
`!=`	Returns `true` if the left operand is not equal to the right operand.
`===`	Returns `true` if the left operand is identical to the right operand. This is not the same as `==`.
`!==`	Returns `true` if the left operand is not identical to the right operand. This is not the same as `!=`.
`<`	Returns `true` if the left operand is smaller than the right operand.
`>`	Returns `true` if the left operand is greater than the right operand.
`<=`	Returns `true` if the left operand is equal to or smaller than the right operand.
`&&`	Returns `true` if both the left operand and the right operand are true.
`||`	Returns `true` if either the left operand or the right operand is true.
`++`	Increments the operand by 1.
`—`	Decrements the operand by 1.
`+=`	Increments the left operand by the right operand.
`-=`	Decrements the left operand by the right operand.
`.`	Concatenates the left operand and the right operand (joins them together).
`%`	Divides the left operand by the right operand and returns the remainder.
`|`	Performs a bitwise `OR` operation. It returns a number with bits that are set in either the left operand or the right operand.

At least 10 other operators are not listed in the table because you're unlikely to use them. Even some of the ones in the table are used infrequently (bitwise `AND`, for example). Having said that, the bitwise `OR` operator (`|`) is used regularly because it allows you to combine values.

Here is a code example that demonstrates some of the operators:

```
<?php
  $i = 100;
  $i++; // $i is now 101
  $i—; // $i is now 100 again
  $i += 10; // $i is 110
  $i = $i / 2; // $i is 55
  $j = $i; // both $j and $i are 55
  $i = $j % 11; // $i is 0
?>
```

The last line uses modulus, which takes some people a little bit of effort to understand. The result of `$i % 11` is `0` because `$i` is set to `55`, and modulus works by dividing the left operand (`55`) by the right operand (`11`) and returning the remainder. 55 divides by 11 exactly 5 times, and so has no remainder, or `0`.

The concatenation operator, a period (`.`), sounds scarier than it is: It just joins strings together. Here is an example:

```
<?php
echo "Hello, " . "world!";
echo "Hello, world!" . "\n";
?>
```

There are two special operators in PHP that are not covered here and yet are used frequently. Before we look at them, though, it is important that you see how the comparison operators (such as `<`, `<=`, and `!=`) are used inside conditional statements.

Conditional Statements

In a *conditional statement*, you instruct PHP to take different actions depending on the outcome of a test. For example, you might want PHP to check whether a variable is greater than 10 and, if so, print a message. This is all done with the `if` statement, which looks like this:

```
if (your condition) {
  // action to take if condition is true
} else {
  // optional action to take otherwise
}
```

The *your condition* part can be filled with any number of conditions you want PHP to evaluate, and this is where the comparison operators come into their own. Consider this example:

```
if ($i > 10) {
  echo "11 or higher";
} else {
  echo "10 or lower";
}
```

PHP looks at the condition and compares `$i` to `10`. If it is greater than 10, it replaces the whole operation with `1`; otherwise, it replaces it with `0`. So, if `$i` is `20`, the result looks like this:

```
if (1) {
  echo "11 or higher";
} else {
  echo "10 or lower";
}
```

In conditional statements, any number other than `0` is considered to be equivalent to the Boolean value `true`; so `1` always evaluates to `true`. There is a similar case for strings: If your string has any characters in it, it evaluates to `true`, with empty strings evaluating to `false`. This is important because you can then use that `1` in another condition through `&&` or `||` operators. For example, if you want to check whether `$i` is greater than `10` but less than `40`, you could write this:

```
if ($i > 10 && $i < 40) {
  echo "11 or higher";
} else {
  echo "10 or lower";
}
```

If you presume that `$i` is set to `50`, the first condition (`$i, 10`) is replaced with `1`, and the second condition (`$i < 40`) is replaced with `0`. Those two numbers are then used by the `&&` operator, which requires both the left and right operands to be `true`. Whereas `1` is equivalent to `true`, `0` is not, so the `&&` operand is replaced with `0`, and the condition fails.

`=`, `==`, `===`, and similar operators are easily confused and often the source of programming errors. The first, a single equal sign, assigns the value of the right operand to the left operand. However, all too often you see code like this:

```
if ($i = 10) {
  echo "The variable is equal to 10!";
} else {
  echo "The variable is not equal to 10";
}
```

This code is incorrect. Rather than checking whether `$i` is equal to `10`, it assigns `10` to `$i` and returns `true`. What is needed is `==`, which compares two values for equality. In PHP, this is extended so that there is also `===` (three equal signs), which checks whether two values are identical, more than just equal.

The difference is slight but important: If you have a variable with the string value `"10"` and compare it against the number value of `10`, they are equal. Thus, PHP converts the type and checks the numbers. However, they are not identical. To be considered identical, the two variables must be equal (that is, have the same value) and be of the same data type (that is, both strings, both integers, and so on).

NOTE

Putting function calls in conditional statements rather than direct comparisons is common practice. Here is an example:

```
if (do_something()) {
```

If the `do_something()` function returns `true` (or something equivalent to `true`, such as a nonzero number), the conditional statement evaluates to `true`.

Special Operators

The ternary operator and the execution operator work differently from the operators you have seen so far. The ternary operator is rarely used in PHP, thankfully, because it is really just a condensed conditional statement. Presumably it arose through someone needing to make code occupy as little space as possible because it certainly does not make PHP code any easier to read.

The ternary operator works like this:

```
$age_description = ($age < 18) ? "child" : "adult";
```

Without explanation, this code is essentially meaningless; however, it expands into the following five lines of code:

```
if ($age < 18) {
  $age_description = "child";
} else {
  $age_description = "adult";
}
```

The ternary operator is so named because it has three operands: a condition to check (`$age < 18` in the previous code), a result if the condition is `true` (`"child"`), and a result if the condition is `false ("adult"`). Although we hope you never have to use the ternary operator, it is at least important to know how it works in case you stumble across it.

The other special operator is the execution operator, which is the backtick symbol (`` ` ``). The position of the backtick key varies depending on your keyboard, but it is likely to be just to the left of the 1 key (above Tab). The execution operator executes the program inside the backticks, returning any text the program outputs. Here is an example:

```
<?php
  $i = `ls -l`;
  echo $i;
?>
```

This executes the `ls` program, passing in `-l` (a lowercase *L*) to get the long format, and stores all its output in `$i`. You can make the command as long or as complex as you like, including piping to other programs. You can also use PHP variables inside the command.

Switching

Having multiple `if` statements in one place is ugly, slow, and prone to errors. Consider the code in Listing 44.3.

LISTING 44.3 How Multiple Conditional Statements Lead to Ugly Code

```
<?php
  $cat_age = 3;

  if ($cat_age == 1) {
    echo "Cat age is 1";
  } else {
    if ($cat_age == 2) {
      echo "Cat age is 2";
    } else {
      if ($cat_age == 3) {
        echo "Cat age is 3";
      } else {
        if ($cat_age == 4) {
          echo "Cat age is 4";
        } else {
          echo "Cat age is unknown";
        }
      }
    }
  }
?>
```

Even though it certainly works, the code in Listing 44.3 is a poor solution to the problem. Much better is a `switch/case` block, which transforms the previous code into what's shown in Listing 44.4.

LISTING 44.4 Using a `switch/case` Block

```
<?php
  $cat_age = 3;

  switch ($cat_age) {
    case 1:
      echo "Cat age is 1";
      break;
    case 2:
      echo "Cat age is 2";
      break;
    case 3:
      echo "Cat age is 3";
      break;
    case 4:
      echo "Cat age is 4";
      break;
    default:
      echo "Cat age is unknown";
  }
?>
```

Although it is only slightly shorter, Listing 44.4 is a great deal more readable and much easier to maintain. A `switch`/`case` group is made up of a `switch()` statement in which you provide the variable you want to check, followed by numerous `case` statements. Notice the `break` statement at the end of each `case`. Without that, PHP would execute each `case` statement beneath the one it matches. Calling `break` causes PHP to exit the `switch`/`case`. Notice also that there is a default `case` at the end that catches everything that has no matching case.

It is important that you not use `case default`: but merely `default`:. Also, it is the last `case` label, so it has no need for a `break` statement because PHP exits the `switch`/`case` block there anyway.

Loops

PHP has four ways you can execute a block of code multiple times: by using `while`, `for`, `foreach`, or `do...while`. Of the four, only `do...while` sees little use; the others are popular, and you will certainly encounter them in other people's scripts.

The most basic loop is the `while` loop, which executes a block of code for as long as a given condition is `true`. So, you can write an infinite loop—a block of code that continues forever—with this PHP:

```
<?php
  $i = 10;
  while ($i >= 10) {
    $i += 1;
    echo $i;
  }
?>
```

The loop block checks whether `$i` is greater or equal to `10` and, if that condition is `true`, adds `1` to `$i` and prints it. Then it goes back to the loop condition again. Because `$i` starts at `10` and you only ever add numbers to it, that loop continues forever. With two small changes, you can make the loop count down from 10 to 0:

```
<?php
  $i = 10;
  while ($i >= 0) {
    $i -= 1;
    echo $i;
  }
?>
```

So, this time you check whether `$i` is greater than or equal to 0 and subtract 1 from it with each loop iteration. You typically use `while` loops when you are unsure how many times the code needs to loop because `while` keeps looping until an external factor stops it.

With a `for` loop, you specify precise limits on its operation by giving it a declaration, a condition, and an action. That is, you specify one or more variables that should be set when the loop first runs (the *declaration*), you set the circumstances that will cause the loop to terminate (the *condition*), and you tell PHP what it should change with each loop iteration (the *action*). That last part is what really sets a `for` loop apart from a `while` loop: You usually tell PHP to change the condition variable with each iteration.

You can rewrite the script that counts down from 10 to 0 by using a `for` loop:

```
<?php
  for($i = 10; $i >= 0; $i -= 1) {
    echo $i;
  }
?>
```

This time you do not need to specify the initial value for `$i` outside the loop, and you also do not need to change `$i` inside the loop; it is all part of the `for` statement. The actual amount of code is really the same, but for this purpose, the `for` loop is arguably tidier and therefore easier to read. With the `while` loop, the `$i` variable was declared outside the loop and so was not explicitly attached to the loop.

The third loop type is `foreach`, which is specifically for arrays and objects, although it is rarely used for anything other than arrays. A `foreach` loop iterates through each element in an array (or each variable in an object), optionally providing both the key name and the value.

In its simplest form, a `foreach` loop looks like this:

```
<?php
  foreach($myarr as $value) {
    echo $value;
  }
?>
```

This loops through the `$myarr` array you created earlier, placing each value in the `$value` variable. You can modify it so you get the keys as well as the values from the array, like this:

```
<?php
  foreach($myarr as $key => $value) {
    echo "$key is set to $value\n";
  }
?>
```

As you can guess, this time the array keys go in `$key`, and the array values go in `$value`. One important characteristic of the `foreach` loop is that it goes from the start of the array to the end and then stops—and by *start* we mean the first item to be added rather than the lowest index number. This script shows this behavior:

```
<?php
  $array = array(6 => "Hello", 4 => "World",
                 2 => "Wom", 0 => "Bat");
  foreach($array as $key => $value) {
    echo "$key is set to $value\n";
  }
?>
```

If you try this script, you will see that `foreach` prints the array in the original order of `6`, `4`, `2`, `0` rather than the numerical order of `0`, `2`, `4`, `6`.

The `do...while loop` works like the `while` loop, with the exception that the condition appears at the end of the code block. This small syntactical difference means a lot, though, because a `do...while` loop is always executed at least once. Consider this script:

```
<?php
  $i = 10;
  do {
    $i -= 1;
    echo $i;
  } while ($i < 10);
?>
```

Without running the script, what do you think it will do? One possibility is that it will do nothing; `$i` is set to `10`, and the condition states that the code must loop only while `$i` is less than `10`. However, a `do...while` loop always executes once, so what happens is that `$i` is set to `10`, and PHP enters the loop, decrements `$i`, prints it, and then checks the condition for the first time. At this point, `$i` is indeed less than `10`, so the code loops, `$i` is decremented again, the condition is rechecked, `$i` is decremented again, and so on. This is, in fact, an infinite loop and so should be avoided!

If you ever want to exit a loop before it has finished, you can use the same `break` statement that you used earlier to exit a `switch/case` block. This becomes more interesting if you find yourself with *nested* loops (loops inside loops). This is a common situation. For example, you might want to loop through all the rows in a chessboard and, for each row, loop through each column. Calling `break` exits only one loop or `switch/case`, but you can use `break 2` to exit two loops or `switch/cases`, or `break 3` to exit three, and so on.

Including Other Files

Unless you are restricting yourself to the simplest programming ventures, you will want to share code among your scripts at some point. The most basic need for this is to have a standard header and footer for your website, with only the body content changing.

However, you might also find yourself with a small set of custom functions you use frequently, and it would be an incredibly bad move to simply copy and paste the functions into each of the scripts that use them.

The most common way to include other files is by using the `include` keyword. Save this script as `include1.php`:

```
<?php
  for($i = 10; $i >= 0; $i -= 1) {
    include "echo_i.php";
  }
?>
```

Then save this script as `echo_i.php`:

```
<?php
  echo $i;
?>
```

If you run `include1.php`, PHP loops from 10 to 0 and includes `echo_i.php` each time. For its part, `echo_i.php` just prints the value of `$i`, which is a crazy way of performing an otherwise simple operation, but it does demonstrate how included files share data. Note that the `include` keyword in `include1.php` is inside a PHP block, but you reopen PHP inside `echo_i.php`. This is important because PHP exits PHP mode for each new file, so you always have a consistent entry point.

Basic Functions

PHP has a vast number of built-in functions that enable you to manipulate strings, connect to databases, and more. There is not room here to cover even 10 percent of the functions. For more detailed coverage of functions, check the "References" section at the end of this chapter.

Strings

Several important functions are used for working with strings, and there are many more of them that are less frequently used that we do not cover here. We look at the most important functions here, ordered by difficulty—easiest first!

The easiest function is `strlen()`, which takes a string as its parameter and returns the number of characters in there, like this:

```
<?php
  $ourstring = "  The Quick Brown Box Jumped Over The Lazy Dog  ";
  echo strlen($ourstring);
?>
```

We use this same string in subsequent examples to save space. If you execute this script, it outputs `48` because 48 characters are in the string. Note the 2 spaces on either side of the text, which pad the 44-character phrase up to 48 characters. You can fix that padding with the `trim()` function, which takes a string to trim and returns it with all the whitespace removed from either side. This is a commonly used function because many strings have an extra new line at the end or a space at the beginning, and `trim()` cleans them up perfectly.

Using `trim()`, you can turn the 48-character string into a 44-character string (the same thing but without the extra spaces) like this:

```
echo trim($ourstring);
```

Keep in mind that `trim()` returns the trimmed string, so it outputs `"The Quick Brown Box Jumped Over The Lazy Dog"`. You can modify it so `trim()` passes its return value to `strlen()` so that the code trims it and then outputs its trimmed length:

```
echo strlen(trim($ourstring));
```

PHP always executes the innermost functions first, so the previous code passes `$ourstring` through `trim()`, uses the return value of `trim()` as the parameter for `strlen()`, and prints it.

Of course, everyone knows that boxes do not jump over dogs; the usual phrase is "the quick brown fox." Fortunately, there is a function to fix that problem: `str_replace()`. Note that it has an underscore in it. (PHP is inconsistent on this matter, so you really need to memorize the function names.)

The `str_replace()` function takes three parameters: the text to search for, the text to replace it with, and the string you want to work with. When working with search functions, people often talk about *needles* and *haystacks*. In this situation, the first parameter is the needle (the thing to find), and the third parameter is the haystack (what you are searching through).

So, you can fix the error and correct *box* to *fox* with this code:

```
echo str_replace("Box", "Fox", $ourstring);
```

There are two little addendums to make here. First, note that we have specified `"Box"` as opposed to `"box"` because that is how it appears in the text. The `str_replace()` function is a *case-sensitive* function, which means it does not consider `"Box"` to be the same as `"box"`. If you want to do a non-case-sensitive search and replace, you can use the `stri_replace()` function, which works the same way but doesn't care about case.

The second addendum is that because you are actually changing only one character (*B* to *F*), you need not use a function at all. PHP enables you to read (and change) individual characters of a string by specifying the character position inside braces (`{}`). As with arrays, strings are zero based, which means in the `$ourstring` variable `$ourstring{0}` is `T`, `$ourstring{1}` is `h`, `$ourstring{2}` is `e`, and so on. You could use this instead of `str_replace()`, like this:

```
<?php
```

```
  $ourstring = "  The Quick Brown Box Jumped Over The Lazy Dog  ";
  $ourstring{18} = "F";
  echo $ourstring;
?>
```

You can extract part of a string by using the `substr()` function, which takes a string as its first parameter, a start position as its second parameter, and an optional length as its third parameter. Optional parameters are common in PHP. If you do not provide them, PHP assumes a default value. In this case, if you specify only the first two parameters, PHP copies from the start position to the end of the string. If you specify the third parameter, PHP copies that many characters from the start. You can write a simple script to print `"Lazy Dog"` by setting the start position to `38`, which, remembering that PHP starts counting string positions from 0, copies from the 39th character to the end of the string:

```
echo substr($ourstring, 38);
```

If you just want to print the word `"Lazy"`, you need to use the optional third parameter to specify the length as `4`, like this:

```
echo substr($ourstring, 38, 4);
```

You can also use the `substr()` function with negative second and third parameters. If you specify just the first and second parameters and provide a negative number for the second parameter, `substr()` counts backward from the end of the string. So, rather than specifying `38` for the second parameter, you can use `-10`, so it takes the last 10 characters from the string. Using a negative second parameter and positive third parameter counts backward from the end string and then uses a forward length. You can print `"Lazy"` by counting 10 characters back from the end and then taking the next 4 characters forward:

```
echo substr($ourstring, -10, 4);
```

Finally, you can use a negative third parameter, too, which also counts back from the end of the string. For example, using `"-4"` as the third parameter means to take everything except the last four characters. Confused yet? This code example should make it clear:

```
echo substr($ourstring, -19, -11);
```

This counts 19 characters backward from the end of the string (which places it at the `O` in `Over`) and then copies everything from there until 11 characters before the end of the string. That prints `Over The`. You could write the same thing using `-19` and `8`, or even `29` and `8`; there is more than one way to do it.

Moving on, the `strpos()` function returns the position of a particular substring inside a string; however, it is most commonly used to answer the question “Does this string contain a specific substring?” You need to pass two parameters to it: a haystack and a needle. (Yes, that’s a different order from `str_replace()`.)

In its most basic use, `strpos()` can find the first instance of `Box` in your phrase, like this:

```
echo strpos($ourstring, "Box");
```

This outputs `18` because that is where the `B` in `Box` starts. If `strpos()` cannot find the substring in the parent string, it returns `false` rather than the position. Much more helpful, though, is the ability to check whether a string contains a substring; a first attempt to check whether your string contains the word `The` might look like this:

```
<?php
  $ourstring = "The Quick Brown Box Jumped Over The Lazy Dog";
  if (strpos($ourstring, "The")) {
    echo "Found 'The'!\n";
  } else {
    echo "'The' not found!\n";
  }
?>
```

Note that we have temporarily taken out the leading and trailing white space from `$ourstring` and are using the return value of `strpos()` for the conditional statement. This reads, "If the string is found, print a message; if not, print another message." Or does it?

Run the script, and you see that it prints the `"not found"` message. The reason for this is that `strpos()` returns `false` if the substring is not found and otherwise returns the position where it starts. Recall that any nonzero number equates to `true` in PHP, which means that `0` equates to `false`. With that in mind, what is the string index of the first `The` in your phrase? Because PHP's strings are zero based, and you no longer have the spaces on either side of the string, the `The` is at position `0`, which your conditional statement evaluates to `false` (hence, the problem).

The solution here is to check for identicality. You know that `0` and `false` are equal, but they are not identical because `0` is an integer, whereas `false` is a Boolean. So, you need to rewrite the conditional statement to see whether the return value from `strpos()` is identical to `false`, and if it is, the substring was not found:

```
<?php
  $ourstring = "The Quick Brown Box Jumped Over The Lazy Dog";
  if (strpos($ourstring, "The") !== false) {
    echo "Found 'The'!\n";
  } else {
    echo "'The' not found!\n";
  }
?>
```

Arrays

Working with arrays is no easy task, but PHP makes it less difficult by providing a selection of functions that can sort, shuffle, intersect, and filter them. (As with other functions, there is only space here to cover a selection; this chapter is by no means a definitive reference to PHP's array functions.)

The easiest array function to use is `array_unique()`, which takes an array as its only parameter and returns the same array with all duplicate values removed. Also in the realm of “so easy you do not need a code example” is the `shuffle()` function, which takes an array as its parameter and randomizes the order of its elements. Note that `shuffle()` does not return the randomized array; it uses your parameter as a reference and scrambles it directly. The last too-easy-to-demonstrate function is `in_array()`, which takes a value as its first parameter and an array as its second and returns `true` if the value is in the array.

With those functions out of the way, we can focus on the more interesting functions, two of which are `array_keys()` and `array_values()`. They both take an array as their only parameter and return a new array made up of the keys in the array or the values of the array, respectively. Using the `array_values()` function is an easy way to create a new array of the same data, just without the keys. This is often used if you have numbered your array keys, deleted several elements, and want to reorder the array.

The `array_keys()` function creates a new array where the values are the keys from the old array, like this:

```
<?php
  $myarr = array("foo" => "red", "bar" => "blue", "baz" => "green");
  $mykeys = array_keys($myarr);
  foreach($mykeys as $key => $value) {
    echo "$key = $value\n";
  }
?>
```

This prints `"0 = foo"`, `"1 = bar"`, and `"2 = baz"`.

Several functions are used specifically for array sorting, but only two get much use: `asort()` and `ksort()`. `asort()` sorts an array by its values, and `ksort()` sorts the array by its keys. Given the array `$myarr` from the previous example, sorting by values would produce an array with elements in the order `bar/blue`, `baz/green`, and `foo/red`. Sorting by key would give the elements in the order `bar/blue`, `baz/green`, and `foo/red`. As with the `shuffle()` function, both `asort()` and `ksort()` do their work *in place*, meaning they return no value but directly alter the parameter you pass in. For interest’s sake, you can also use `arsort()` and `krsort()` for reverse value sorting and reverse key sorting, respectively.

This code example reverse sorts the array by value and then prints it, as before:

```
<?php
  $myarr = array("foo" => "red", "bar" => "blue", "baz" => "green");
  arsort($myarr);
  foreach($myarr as $key => $value) {
    echo "$key = $value\n";
  }
?>
```

Previously, when discussing constants, we mentioned the `extract()` function, which converts an array into individual variables; now it is time to start using it for real. You need to provide three variables: the array you want to extract, how you want the variables prefixed, and the prefix you want used. Technically, the last two parameters are optional, but practically you should always use them to properly namespace your variables and keep them organized.

The second parameter must be one of the following:

- **`EXTR_OVERWRITE`**—If the variable exists already, overwrites it.
- **`EXTR_SKIP`**—If the variable exists already, skips it and moves on to the next variable.
- **`EXTR_PREFIX_SAME`**—If the variable exists already, uses the prefix specified in the third parameter.
- **`EXTR_PREFIX_ALL`**—Prefixes all variables with the prefix in the third parameter, regardless of whether it exists already.
- **`EXTR_PREFIX_INVALID`**—Uses a prefix only if the variable name would be invalid (for example, starting with a number).
- **`EXTR_IF_EXISTS`**—Extracts only variables that already exist. We have never seen this used.

You can also, optionally, use the bitwise `OR` operator, `|`, to add in `EXTR_REFS` to have `extract()` use references for the extracted variables. In general use, `EXTR_PREFIX_ALL` is preferred because it guarantees namespacing. `EXTR_REFS` is required only if you need to be able to change the variables and have those changes reflected in the array.

This next script uses `extract()` to convert `$myarr` into individual variables, `$arr_foo`, `$arr_bar`, and `$arr_baz`:

```
<?php
  $myarr = array("foo" => "red", "bar" => "blue", "baz" => "green");
  extract($myarr, EXTR_PREFIX_ALL, 'arr');
?>
```

Note that the array keys are `"foo"`, `"bar"`, and `"baz"` and that the prefix is `"arr"`, but that the final variables will be `$arr_foo`, `$arr_bar`, and `$arr_baz`. PHP inserts an underscore between the prefix and array key.

Files

As you have learned elsewhere in the book, the UNIX philosophy is that everything is a file. In PHP, this is also the case: A selection of basic file functions is suitable for opening and manipulating files, but those same functions can also be used for opening and manipulating network sockets. We cover both here.

Two basic read and write functions for files make performing these basic operations easy. They are `file_get_contents()`, which takes a filename as its only parameter and returns the file's contents as a string, and `file_put_contents()`, which takes a filename as its first parameter and the data to write as its second parameter.

Using these two functions, you can write a script that reads all the text from one file, `filea.txt`, and writes it to another, `fileb.txt`:

```
<?php
  $text = file_get_contents("filea.txt");
  file_put_contents("fileb.txt", $text);
?>
```

Because PHP enables you to treat network sockets like files, you can also use `file_get_contents()` to read text from a website, like this:

```
<?php
  $text = file_get_contents("http://www.slashdot.org");
  file_put_contents("fileb.txt", $text);
?>
```

The problem with using `file_get_contents()` is that it loads the whole file into memory at once; that's not practical if you have large files or even smaller files being accessed by many users. An alternative is to load the file piece by piece, which can be accomplished by using the following five functions: `fopen()`, `fclose()`, `fread()`, `fwrite()`, and `feof()`. The `f` in these function names stands for *file*, so they open, close, read from, and write to files and sockets. The last function, `feof()`, returns `true` if the end of the file has been reached.

On the surface, the `fopen()` function looks straightforward, though it takes a bit of learning to use properly. Its first parameter is the filename you want to open, which is easy enough. However, the second parameter is where you specify how you want to work with the file, and you should specify one of the following:

- **`r`**—Read-only; it overwrites the file.
- **`r+`**—Reading and writing; it overwrites the file.
- **`w`**—Write-only; it erases the existing contents and overwrites the file.
- **`w+`**—Reading and writing; it erases the existing content and overwrites the file.
- **`a`**—Write-only; it appends to the file.
- **`a+`**—Reading and writing; it appends to the file.
- **`x`**—Write-only, but only if the file does not exist.
- **`a+`**—Reading and writing, but only if the file does not exist.

Optionally, you can also add `b` (for example, `a+b` or `rb`) to switch to binary mode. This is recommended if you want your scripts and the files they write to work smoothly on other platforms.

When you call `fopen()`, you should store the return value. It is a resource known as a *file handle*, which the other file functions all need to do their jobs. The `fread()` function, for example, takes the file handle as its first parameter and the number of bytes to read as its second, and it returns the content in its return value. The `fclose()` function takes the file handle as its only parameter and frees up the file.

So, you can write a simple loop to open a file, read it piece by piece, print the pieces, and then close the handle:

```
<?php
  $file = fopen("filea.txt", "rb");
  while (!feof($file)) {
    $content = fread($file, 1024);
    echo $content;
  }
  fclose($file);
?>
```

This leaves only the `fwrite()` function, which takes the file handle as its first parameter and the string to write as its second. You can also provide an integer as the third parameter, specifying the number of bytes you want to write of the string, but if you exclude this, `fwrite()` writes the entire string.

Recall that you can use `a` as the second parameter to `fopen()` to append data to a file. So, you can combine that with `fwrite()` to have a script that adds a line of text to a file each time it is executed:

```
<?php
  $file = fopen("filea.txt", "ab");
  fwrite($file, "Testing\n");
  fclose($file);
?>
```

To make this script a little more exciting, you can stir in a new function, `filesize()`, that takes a filename (not a file handle, but an actual filename string) as its only parameter and returns the file's size, in bytes. Using the `filesize()` function brings the script to this:

```
<?php
  $file = fopen("filea.txt", "ab");
  fwrite($file, "The filesize was" . filesize("filea.txt") . "\n");
  fclose($file);
?>
```

Although PHP automatically cleans up file handles for you, it is still best to use `fclose()` yourself so that you are always in control.

Miscellaneous

Several functions do not fall under the other categories and so are covered here. The first one is `isset()`, which takes one or more variables as its parameters and returns `true` if they have been set. It is important to note that a variable with a value set to something that would be evaluated to `false`—such as `0` or an empty string—still returns `true` from `isset()` because this function does not check the value of the variable. It merely checks that it is set; hence, the name.

The `unset()` function also takes one or more variables as its parameters, simply deleting the variable(s) and freeing up the memory. With `isset()` and `unset()`, you can write a script that checks for the existence of a variable and, if it exists, deletes it (see Listing 44.5).

LISTING 44.5 Setting and Unsetting Variables

```
<?php
  $name = "Ildiko";
  if (isset($name)) {
    echo "Name was set to $name\n";
    unset($name);
  } else {
    echo "Name was not set";
  }

  if (isset($name)) {
    echo "Name was set to $name\n";
    unset($name);
  } else {
    echo "Name was not set";
  }
?>
```

This script runs the same `isset()` check twice, but it uses `unset()` on the variable after the first check. It therefore prints `"Name was set to Ildiko"` and then `"Name was not set"`.

Perhaps the most frequently used function in PHP is `exit`, although purists will tell you that it is, in fact, a language construct rather than a function. `exit` terminates the processing of the script as soon as it is executed, which means subsequent lines of code are not executed. That is really all there is to it; it barely deserves an example, but here is one just to make sure:

```
<?php
  exit;
  echo "Exit is a language construct!\n";
?>
```

This script prints nothing because the `exit` comes before the `echo`.

One function we can guarantee you will use a lot is `var_dump()`, which dumps out information about a variable, including its value, to the screen. This function is invaluable for arrays because it prints every value and, if one or more of the elements is an array, it prints all the elements from those, and so on. To use this function, just pass a variable to it as its only parameter, as shown here:

```
<?php
  $drones = array("Graham", "Julian", "Nick", "Paul");
  var_dump($drones);
?>
```

The output from this script looks as follows:

```
array(4) {
  [0]=>
  string(6) "Graham"
  [1]=>
  string(6) "Julian"
  [2]=>
  string(4) "Nick"
  [3]=>
  string(4) "Paul"
}
```

The `var_dump()` function sees a lot of use as a basic debugging technique because using it is the easiest way to print variable data to the screen to verify it.

Finally, we briefly discuss regular expressions—with the emphasis on *briefly* because regular expression syntax is covered elsewhere in this book, and the only unique thing relevant to PHP is the functions you use to run the expressions. You have the choice of either *Perl-Compatible Regular Expressions* (*PCRE*) or POSIX Extended regular expressions, but there really is little to choose between them in terms of functionality offered. For this chapter, we use the PCRE expressions because, to the best of our knowledge, they see more use by other PHP programmers.

The main PCRE functions are `preg_match()`, `preg_match_all()`, `preg_replace()`, and `preg_split()`. We start with `preg_match()` because it provides the most basic functionality, returning `true` if one string matches a regular expression. The first parameter to `preg_match()` is the regular expression you want to search for, and the second is the string to match. So, if you want to check whether a string has the word `Best`, `Test`, `rest`, `zest`, or any other word containing `est` preceded by any letter of either case, you could use this PHP code:

```
$result = preg_match("/[A-Za-z]est/", "This is a test");
```

Because the test string matches the expression, `$result` is set to `1` (`true`). If you change the string to a nonmatching result, you get `0` as the return value.

The next function is `preg_match_all()`, which gives you an array of all the matches it found. However, to be most useful, it takes the array to fill with matches as a by reference parameter and saves its return value for the number of matches that were found.

We suggest that you use `preg_match_all()` and `var_dump()` to get a feel for how the `preg_match_all()` function works. This example is a good place to start:

```
<?php
  $string = "This is the best test in the west";
  $result = preg_match_all("/[A-Za-z]est/", $string, $matches);
  var_dump($matches);
?>
```

It outputs the following:

```
array(1) {
  [0]=>
  array(3) {
    [0]=>
    string(4) "best"
    [1]=>
    string(4) "test"
    [2]=>
    string(4) "west"
  }
}
```

Notice that the `$matches` array is actually multidimensional in that it contains one element, which itself is an array containing all the matches to the regular expression. This is because the expression has no *subexpressions*, meaning no independent matches using parentheses. If you had subexpressions, each would have its own element in the `$matches` array, containing its own array of matches.

Moving on, `preg_replace()` is used to change all substrings that match a regular expression into something else. The basic manner of using this is quite easy: You search for something with a regular expression and provide a replacement for it. However, a more useful variant is *back referencing*, using the match as part of the replacement. For the sake of this example, say that you have written a tutorial on PHP but want to process the text so each reference to a function is followed by a link to the PHP manual.

PHP manual page URLs take the form www.php.net/<*somefunc*> (for example, www.php.net/preg_replace). The string you need to match is a function name, which is a string of alphabetic characters, potentially also mixed with numbers and underscores and terminated with parentheses, `()`. As a replacement, you can use the match you found, surrounded in HTML emphasis tags (`<em></em>`), and then a link to the relevant PHP manual page. Here is how all this looks in code:

```
<?php
  $regex = "/([A-Za-z0-9_]*)\(\)/";
  $replace = "<em>$1</em> (<a href=\"http://www.php.net/$1\">manual</A>)";
  $haystack = "File_get_contents()is easier than using fopen().";
  $result = preg_replace($regex, $replace, $haystack);
  echo $result;
?>
```

The `$1` is the back reference; it will be substituted with the results from the first subexpression. The way we have written the regular expression is very exact. The `[A-Za-z0-9_]*` part, which matches the function name, is marked as a subexpression. After that is `\(\)`, which means the exact symbols (and), not the regular expression meanings of them, which means that `$1` in the replacement will contain `fopen` rather than `fopen()`, which is how it should be. Of course, anything that is not back referenced in the replacement is removed, so you have to put the `()` after the first `$1` (not in the hyperlink) to repair the function name.

After all that work, the output is perfect:

```
<em>File_get_contents()</em> (<a href="http://www.php.net/
file_get_contents">manual</A>) is easier than using <em>fopen()
</em> (<a href="http://www.php.net/fopen">manual</A>).
```

Handling HTML Forms

Given that PHP's primary role is handling web pages, you might wonder why this section has been left until so late in the chapter. It is because handling HTML forms is so central to PHP that it is essentially automatic.

Consider this form:

```
<form method="POST" action="thispage.php">
User ID: <input type="text" name="UserID" /><br />
Password: <input type="password" name="Password" /><br />
<input type="submit" />
</form>
```

When a visitor clicks Submit, `thispage.php` is called again, and this time PHP has the variables available to it inside the `$_REQUEST` array. Given that script, if the user enters `12345` and `frosties` as her user ID and password, PHP provides `$_REQUEST['UserID']` set to `12345` and `$_REQUEST['Password']` set to `frosties`. Note that it is important that you use HTTP `POST` unless you specifically want `GET`. `POST` enables you to send a great deal more data and stops people from tampering with your URL to try to find holes in your script.

Is that it? Well, almost. You now know how to retrieve user data, but you should be sure to sanitize it so users do not try to sneak HTML or JavaScript into your database as something you think is innocuous. PHP gives you the `strip_tags()` function for this purpose. It takes a string and returns the same string with all HTML tags removed.

Databases

The ease with which PHP can be used to create dynamic, database-driven websites is the key reason to use it for many people. The stock build of PHP comes with support for MySQL, PostgreSQL, SQLite, Oracle, Microsoft SQL Server, ODBC, plus several other popular databases, so you are sure to find something to work with your data.

If you want to, you can learn all the individual functions for connecting to and manipulating each database PHP supports, but a much smarter, or at least easier, idea is to use `PEAR::DB`, which is an abstraction layer over the databases that PHP supports. You write your code once, and—with the smallest of changes—it works on every database server.

PEAR is the script repository for PHP, and it contains numerous tools and prewritten solutions for common problems. `PEAR::DB` is perhaps the most popular part of the PEAR project, but it is worth checking out the PEAR site (https://pear.php.net) to see whether anything else catches your eye.

To get basic use out of `PEAR::DB`, you need to learn how to connect to a database, run an SQL query, and work with the results. This is not an SQL tutorial, so we have assumed that you are already familiar with the language. For the sake of this tutorial, we have also assumed that you are working with a database called `dentists` and a table called `patients` that contains the following fields:

- `ID`—The primary key, an auto-incrementing integer for storing a number unique to each patient
- `Name`—A `varchar(255)` field for storing a patient name
- `Age`—An integer
- `Sex`—`1` for male, `2` for female
- `Occupation`—A `varchar(255)` field for storing a patient occupation

Also for the sake of this tutorial, we use a database server at IP address `10.0.0.1`, running MySQL, with username `ubuntu` and password `alm65z`. You need to replace these details with your own; use `localhost` for connecting to the local server.

The first step to using `PEAR::DB` is to include the standard `PEAR::DB` file, `DB.php`. Your PHP will be configured to look inside the PEAR directory for `include()` files, so you do not need to provide any directory information.

`PEAR::DB` is object oriented, and you specify your connection details at the same time as you create the initial DB object. This is done using a URL-like system that specifies the database server type, username, password, server, and database name all in one. After you have specified the database server here, everything else is abstracted, meaning you only need to change the connection line to port your code to another database server.

This script in Listing 44.6 connects to our server and prints a status message.

LISTING 44.6 Connecting to a Database Through `PEAR::DB`

```
<?php
  include("DB.php");
  $dsn = "mysql://ubuntu:alm65z@10.0.0.1/dentists";
  $conn = DB::connect($dsn);
  if (DB::isError($conn)) {
    echo $conn->getMessage() . "\n";
  } else {
    echo "Connected successfully!\n";

  }
?>
```

You should be able to see how the connection string breaks down. It is server name first, then a username and password separated by a colon, then an `@` symbol followed by the IP address to which to connect, and then a slash and the database name. Notice how the call to connect is `DB::connect()`, which calls `PEAR::DB` directly and returns a database connection object for storage in `$conn`. The variable name `$dsn` was used for the connection details because it is a common acronym that stands for data source name.

If `DB::connect()` successfully connects to a server, it returns a database object you can use to run SQL queries. If not, you get an error returned that you can query using functions such as `getMessage()`. The script in Listing 44.6 prints the error message if the connection fails, but it also prints a message on success. Next, you change that so you run an SQL query if we have a connection.

Running SQL queries is done through the `query()` function of the database connection, passing in the SQL you want to execute. This then returns a query result that can be used to get the data. This query result can be thought of as a multidimensional array because it has many rows of data, each with many columns of attributes. This is extracted using the `fetchInto()` function, which loops through the query result, converting one row of data into an array that it sends back as its return value. You need to pass in two parameters to `fetchInto()` to specify where the data should be stored and how you want it stored. Unless you have unusual needs, specifying `DB_FETCHMODE_ASSOC` for the second parameter is a smart move.

Listing 44.7 shows the new script.

LISTING 44.7 Running a Query Through `PEAR::DB`

```
<?php
  include("DB.php");
  $dsn = "mysql://ubuntu:alm65z@10.0.0.1/dentists";
  $conn = DB::connect($dsn);
  if (DB::isError($conn)) {
    echo $conn->getMessage() . "\n";
  } else {
    echo "Connected successfully!\n";
    $result = $conn->query("SELECT ID, Name FROM patients;");
    while ($result->fetchInto($row, DB_FETCHMODE_ASSOC)) {
      extract($row, EXTR_PREFIX_ALL, 'pat');
      echo "$pat_ID is $pat_Name\n";
    }
  }
?>
```

The first half of Listing 44.7 is identical to the script in Listing 44.6, with all the new action happening if a successful connection occurs.

Going along with the saying "never leave to PHP what you can clean up yourself," the current script has problems. It does not clean up the query result, and it does not close the database connection. If this code were being used in a longer script that ran for several minutes, this would be a huge waste of resources. Fortunately, you can free up the memory associated with these two issues by calling `$result->free()` and `$conn->disconnect()`. If you add those two function calls to the end of the script, it is complete.

References

- **https://secure.php.net**—The best place to look for information is the PHP online manual. It is comprehensive, well written, and updated regularly.
- **www.phpbuilder.com**—This is a large PHP scripts and tutorials site where you can learn new techniques and also chat with other PHP developers.
- **www.zend.com**—This is the home page of a company founded by two of the key developers of PHP. Zend develops and sells proprietary software, including a powerful IDE and a code cache, to aid PHP developers.
- **https://pear.php.net**—The home of the PEAR project contains a large collection of software you can download and try, and it has thorough documentation for it all.
- **www.phparch.com**—Quite a few good PHP magazines are around, but *PHP Architect* leads the way. It posts some of its articles online for free, and its forums are good, too.
- ***PHP and MySQL Web Development* by Luke Welling and Laura Thomson**—This book is the best choice for beginning developers.
- ***PHP in a Nutshell* by Paul Hudson**—This concise, to-the-point book covers all aspects of PHP.
- ***Advanced PHP Programming* by George Schlossnagle**—As its title says, this book is for advanced developers.