2 Scripting and the Shell

Good system administrators write scripts. Scripts standardize and automate the performance of administrative chores and free up admins' time for more important and more interesting tasks. In a sense, scripts are also a kind of low-rent documentation in that they act as an authoritative outline of the steps needed to complete a particular task.

In terms of complexity, administrative scripts run the gamut from simple ones that encapsulate a few static commands to major software projects that manage host configurations and administrative data for an entire site. In this book we're primarily interested in the smaller, day-to-day scripting projects that sysadmins normally encounter, so we don't talk much about the support functions (e.g., bug tracking and design review) that are needed for larger projects.

Administrative scripts should emphasize programmer efficiency and code clarity rather than computational efficiency. This is not an excuse to be sloppy, but simply a recognition that it rarely matters whether a script runs in half a second or two seconds. Optimization can have an amazingly low return on investment, even for scripts that run regularly out of **cron**.

For a long time, the standard language for administrative scripts was the one defined by the shell. Most systems' default shell is **bash** (the "Bourne-again" shell), but **sh** (the original Bourne shell) and **ksh** (the Korn shell) are used on a few UNIX systems. Shell scripts are typically used for light tasks such as automating a sequence of commands or assembling several filters to process data.

The shell is always available, so shell scripts are relatively portable and have few dependencies other than the commands they invoke. Whether or not you choose the shell, the shell may choose you: most environments include a hefty complement of existing **sh** scripts, and those scripts frequently need to be read, understood, and tweaked by administrators.

For more sophisticated scripts, it's advisable to jump to a real programming language such as Perl or Python, both of which are well suited for administrative work. These languages incorporate a couple of decades' worth of language design advancements relative to the shell, and their text processing facilities (invaluable to administrators) are so powerful that **sh** can only weep and cower in shame.

The main drawback to Perl and Python is that their environments can be a bit fussy to set up, especially when you start to use third-party libraries that have compiled components. The shell skirts this particular issue by having no module structure and no third-party libraries.

This chapter takes a quick look at **bash**, Perl, and Python as languages for scripting, along with regular expressions as a general technology.

2.1 Shell basics

Before we take up shell scripting, let's review some of the basic features and syntax of the shell. The material in this section applies to all major shells in the **sh** lineage (including **bash** and **ksh**, but not **csh** or **tcsh**), regardless of the exact platform you are using. Try out the forms you're not familiar with, and experiment!

Command editing

We've watched way too many people edit command lines with the arrow keys. You wouldn't do that in your text editor, right?

If you like **emacs**, all the basic **emacs** commands are available to you when you're editing history. <Control-E> goes to the end of the line and <Control-A> to the beginning. <Control-P> steps backward through recently executed commands and recalls them for editing. <Control-R> searches incrementally through your history to find old commands.

If you like **vi**, put your shell's command-line editing into **vi** mode like this:

```
$ set -o vi
```

As in **vi**, editing is modal; however, you start in input mode. Type <Esc> to leave input mode and "i" to reenter it. In edit mode, "w" takes you forward a word, "fX" finds the next X in the line, and so on. You can walk through past command history entries with <Esc> k. Want **emacs** editing mode back again?

```
$ set -o emacs
```

Pipes and redirection

Every process has at least three communication channels available to it: “standard input” (STDIN), “standard output” (STDOUT), and “standard error” (STDERR). The kernel sets up these channels on the process’s behalf, so the process itself doesn’t necessarily know where they lead. They might connect to a terminal window, a file, a network connection, or a channel belonging to another process, to name a few possibilities.

UNIX has a unified I/O model in which each channel is named with a small integer called a file descriptor. The exact number assigned to a channel is not usually significant, but STDIN, STDOUT, and STDERR are guaranteed to correspond to file descriptors 0, 1, and 2, so it’s safe to refer to these channels by number. In the context of an interactive terminal window, STDIN normally reads from the keyboard and both STDOUT and STDERR write their output to the screen.

Most commands accept their input from STDIN and write their output to STDOUT. They write error messages to STDERR. This convention lets you string commands together like building blocks to create composite pipelines.

The shell interprets the symbols <, >, and >> as instructions to reroute a command’s input or output to or from a file. A < symbol connects the command’s STDIN to the contents of an existing file. The > and >> symbols redirect STDOUT; > replaces the file’s existing contents, and >> appends to them. For example, the command

```
$ echo "This is a test message." > /tmp/mymessage
```

stores a single line in the file **/tmp/mymessage**, creating the file if necessary. The command below emails the contents of that file to user johndoe.

```
$ mail -s "Mail test" johndoe < /tmp/mymessage
```

To redirect both STDOUT and STDERR to the same place, use the **>&** symbol. To redirect STDERR only, use **2>**.

The **find** command illustrates why you might want to handle STDOUT and STDERR separately because it tends to produce output on both channels, especially when run as an unprivileged user. For example, a command such as

```
$ find / -name core
```

usually results in so many “permission denied” error messages that genuine hits get lost in the clutter. To discard all the error messages, you can use

```
$ find / -name core 2> /dev/null
```

In this version, only real matches (where the user has read permission on the parent directory) come to the terminal window. To save the list of matching paths to a file, try

```
$ find / -name core > /tmp/corefiles 2> /dev/null
```

This command line sends matching paths to **/tmp/corefiles**, discards errors, and sends nothing to the terminal window.

To connect the STDOUT of one command to the STDIN of another, use the **|** symbol, commonly known as a pipe. Some examples:

```
$ ps -ef | grep httpd
$ cut -d: -f7 < /etc/passwd | sort -u
```

The first example runs **ps** to generate a list of processes and pipes it through the **grep** command to select lines that contain the word **httpd**. The output of **grep** is not redirected, so the matching lines come to the terminal window.

The **cut** command in the second example picks out the path to each user's shell from **/etc/passwd**. The list of shells is then sent through **sort -u** to produce a sorted list of unique values.

To execute a second command only if its precursor completes successfully, you can separate the commands with an **&&** symbol. For example,

```
$ lpr /tmp/t2 && rm /tmp/t2
```

removes **/tmp/t2** if and only if it is successfully queued for printing. Here, the success of the **lpr** command is defined as its yielding an exit code of zero, so the use of a symbol that suggests "logical AND" for this purpose may be confusing if you're used to short-circuit evaluation in other programming languages. Don't think about it too much; just accept it as a shell idiom.

Conversely, the || symbol executes the following command only if the preceding command fails (produces a nonzero exit status).

In a script, you can use a backslash to break a command onto multiple lines, helping to distinguish the error-handling code from the rest of the command pipeline:

```
cp --preserve --recursive /etc/* /spare/backup \
    || echo "Did NOT make backup"
```

For the converse effect—multiple commands combined onto one line—you can use a semicolon as a statement separator.

Variables and quoting

Variable names are unmarked in assignments but prefixed with a dollar sign when their values are referenced. For example:

```
$ etcdir='/etc'
$ echo $etcdir
/etc
```

Do not put spaces around the = symbol or the shell will mistake your variable name for a command name.

When referencing a variable, you can surround its name with curly braces to clarify to the parser and to human readers where the variable name stops and other text begins; for example, **${etcdir}** instead of just **$etcdir**. The braces are not normally required, but they can be useful when you want to expand variables inside double-quoted strings. Often, you'll want the contents of a variable to be followed by literal letters or punctuation. For example,

```
$ echo "Saved ${rev}th version of mdadm.conf."
Saved 8th version of mdadm.conf.
```

There's no standard convention for the naming of shell variables, but all-caps names typically suggest environment variables or variables read from global configuration files. More often than not, local variables are all-lowercase with components separated by underscores. Variable names are case sensitive.

Environment variables are automatically imported into **bash**'s variable namespace, so they can be set and read with the standard syntax. Use **export** *varname* to promote a shell variable to an environment variable. Commands for environment variables that you want to set up at login time should be included in your **~/.profile** or **~/.bash_profile** file. Other environment variables, such as PWD for the current working directory, are maintained automatically by the shell.

The shell treats strings enclosed in single and double quotes similarly, except that double-quoted strings are subject to globbing (the expansion of filename-matching metacharacters such as * and ?) and variable expansion. For example:

```
$ mylang="Pennsylvania Dutch"
$ echo "I speak ${mylang}."
I speak Pennsylvania Dutch.
$ echo 'I speak ${mylang}.'
I speak ${mylang}.
```

Back quotes, also known as back-ticks, are treated similarly to double quotes, but they have the additional effect of executing the contents of the string as a shell command and replacing the string with the command's output. For example,

```
$ echo "There are `wc -l /etc/passwd` lines in the passwd file."
There are 28 lines in the passwd file.
```

Common filter commands

Any well-behaved command that reads STDIN and writes STDOUT can be used as a filter (that is, a component of a pipeline) to process data. In this section we briefly review some of the more widely used filter commands (including some used in passing above), but the list is practically endless. Filter commands are so team oriented that it's sometimes hard to show their use in isolation.

Most filter commands accept one or more filenames on the command line. Only if you fail to specify a file do they read their standard input.

cut: separate lines into fields

The **cut** command prints selected portions of its input lines. It's most commonly used to extract delimited fields, as in the example on page 32, but it can return segments defined by column boundaries as well. The default delimiter is <Tab>, but you can change delimiters with the **-d** option. The **-f** options specifies which fields to include in the output.

For an example of the use of **cut**, see the section on **uniq**, below.

sort: sort lines

sort sorts its input lines. Simple, right? Well, maybe not—there are a few potential subtleties regarding the exact parts of each line that are sorted (the "keys") and the collation order to be imposed. Table 2.1 shows a few of the more common options, but check the man page for others.

Table 2.1 sort options

Opt	Meaning
-b	Ignore leading whitespace
-f	Case insensitive sorting
-k	Specify the columns that form the sort key
-n	Compare fields as integer numbers
-r	Reverse sort order
-t	Set field separator (the default is whitespace)
-u	Output unique records only

The commands below illustrate the difference between numeric and dictionary sorting, which is the default. Both commands use the **-t:** and **-k3,3** options to sort the **/etc/group** file by its third colon-separated field, the group ID. The first sorts numerically and the second alphabetically.

```
$ sort -t: -k3,3 -n /etc/group[1]
root:x:0:
bin:x:1:daemon
daemon:x:2:
…

$ sort -t: -k3,3 /etc/group
root:x:0:
bin:x:1:daemon
users:x:100:
…
```

1. **sort** accepts the key specification **-k3** (rather than **-k3,3**), but it probably doesn't do what you expect. Without the terminating field number, the sort key continues to the end of the line.

uniq: print unique lines

uniq is similar in spirit to **sort -u**, but it has some useful options that **sort** does not emulate: **-c** to count the number of instances of each line, **-d** to show only duplicated lines, and **-u** to show only nonduplicated lines. The input must be presorted, usually by being run through **sort**.

For example, the command below shows that 20 users have **/bin/bash** as their login shell and that 12 have **/bin/false**. (The latter are either pseudo-users or users whose accounts have been disabled.)

```
$ cut -d: -f7 /etc/passwd | sort | uniq -c
   20 /bin/bash
   12 /bin/false
```

wc: count lines, words, and characters

Counting the number of lines, words, and characters in a file is another common operation, and the **wc** (word count) command is a convenient way of doing this. Run without options, it displays all three counts:

```
$ wc /etc/passwd
 32  77 2003 /etc/passwd
```

In the context of scripting, it is more common to supply a **-l**, **-w**, or **-c** option to make **wc**'s output consist of a single number. This form is most commonly seen inside backquotes so that the result can be saved or acted upon.

tee: copy input to two places

A command pipeline is typically linear, but it's often helpful to tap into the data stream and send a copy to a file or to the terminal window. You can do this with the **tee** command, which sends its standard input both to standard out and to a file that you specify on the command line. Think of it as a tee fixture in plumbing.

The device **/dev/tty** is a synonym for the current terminal. For example,

```
$ find / -name core | tee /dev/tty | wc -l
```

prints both the pathnames of files named **core** and a count of the number of **core** files that were found.

A common idiom is to terminate a pipeline that will take a long time to run with a **tee** command so that output goes both to a file and to the terminal window for inspection. You can preview the initial results to make sure everything is working as you expected, then leave while the command runs, knowing that the results will be saved.

***head** and **tail**: read the beginning or end of a file*

Reviewing lines from the beginning or end of a file is a common administrative operation. These commands display ten lines by default, but you can include a command-line option to specify how many lines you want to see.

For interactive use, **head** is more or less obsoleted by the **less** command, which paginates files for display. But **head** still finds plenty of use within scripts.

tail also has a nifty **-f** option that's particularly useful for sysadmins. Instead of exiting immediately after printing the requested number of lines, **tail -f** waits for new lines to be added to the end of the file and prints them as they appear— great for monitoring log files. Be aware, however, that the program writing the file may be buffering its output. Even if lines are being added at regular intervals from a logical perspective, they may only become visible in chunks of 1KiB or 4KiB.[2]

Type <Control-C> to stop monitoring.

***grep**: search text*

grep searches its input text and prints the lines that match a given pattern. Its name is based on the **g/***regular-expression***/p** command from the old **ed** editor that came with the earliest versions of UNIX (and still does).

"Regular expressions" are text-matching patterns written in a standard and well-characterized pattern matching language. They're a universal standard used by most programs that do pattern matching, although there are minor variations among implementations. The odd name stems from regular expressions' sordid origins in theory-of-computation studies. We discuss regular expression syntax in more detail starting on page 48.

Like most filters, **grep** has many options, including **-c** to print a count of matching lines, **-i** to ignore case when matching, and **-v** to print nonmatching (rather than matching) lines. Another useful option is **-l** (lowercase L), which makes **grep** print only the names of matching files rather than printing each line that matches. For example, the command

```
$ sudo grep -l mdadm /var/log/*
/var/log/auth.log
/var/log/syslog.0
```

shows that log entries from **mdadm** have appeared in two different log files.

grep is traditionally a fairly basic regular expression engine, but some versions permit the selection of other dialects. For example, **grep -p** on Linux selects Perl-style expressions, though the man page warns darkly that they are "highly experimental." If you need full power, just use Perl or Python.

2. See *Units* on page 14 for an introduction to these units.

2.2 BASH SCRIPTING

bash is great for simple scripts that automate things you'd otherwise be typing on the command line. Your command-line skills carry over to **bash** scripting, and vice versa, which helps you extract maximum value from the learning time you invest in **bash**. But once a **bash** script gets above a hundred lines or you need features that **bash** doesn't have, it's time to move on to Perl or Python.

bash comments start with a hash mark (#) and continue to the end of the line. As on the command line, you can break a single logical line onto multiple physical lines by escaping the newline with a backslash. You can also put more than one statement on a line by separating the statements with semicolons.

A **bash** script may consist of nothing but a series of command lines. For example, the following **helloworld** script simply does an **echo**.

```
#!/bin/bash
echo "Hello, world!"
```

The first line is known as the "shebang" statement and declares the text file to be a script for interpretation by **/bin/bash**. The kernel looks for this syntax when deciding how to execute the file. From the perspective of the shell spawned to execute the script, the shebang line is just a comment. If **bash** were in a different location, you would need to adjust this line.

To prepare the file for running, just turn on its execute bit (see page 156).

```
$ chmod +x helloworld
$ ./helloworld[3]
Hello, world!
```

You can also invoke the shell as an interpreter directly:

```
$ bash helloworld
Hello, world!
$ source helloworld
Hello, world!
```

The first command runs **helloworld** in a new instance of **bash**, and the second makes your existing login shell read and execute the contents of the file. The latter option is useful when the script sets up environment variables or makes other customizations that apply only to the current shell. It's commonly used in scripting to incorporate the contents of a configuration file written as a series of **bash** variable assignments.[4]

3. If your shell understands the command **helloworld** without the **./** prefix, that means the current directory (**.**) is in your search path. This is bad because it gives other users the opportunity to lay traps for you in the hope that you'll try to execute certain commands while **cd**'ed to a directory on which they have write access.
4. The "dot" command is a synonym for **source**, e.g., **. helloworld**.

See page 152 for more information about permission bits.

If you come from the Windows world, you may be accustomed to a file's extension indicating what type of file it is and whether it can be executed. In UNIX and Linux, the file permission bits indicate whether a file can be executed, and if so, by whom. If you wish, you can give your bash scripts a **.sh** suffix to remind you what they are, but you'll then have to type out the **.sh** when you run the command, since UNIX doesn't treat extensions specially.

From commands to scripts

Before we jump into **bash**'s scripting features, a note about methodology. Most people write **bash** scripts the same way they write Perl or Python scripts: with a text editor. However, it's more productive to think of your regular shell command prompt as an interactive script development environment.

For example, suppose you have log files named with the suffixes **.log** and **.LOG** scattered throughout a directory hierarchy and that you want to change them all to the uppercase form. First, let's see if we can find all the files.

```
$ find . -name '*log'
.do-not-touch/important.log
admin.com-log/
foo.log
genius/spew.log
leather_flog
...
```

Oops, it looks like we need to include the dot in our pattern and to leave out directories as well. Type <Control-P> to recall the command and then modify it.

```
$ find . -type f -name '*.log'
.do-not-touch/important.log
foo.log
genius/spew.log
...
```

OK, this looks better. That **.do-not-touch** directory looks dangerous, though; we probably shouldn't mess around in there.

```
$ find . -type f -name '*.log' | grep -v .do-not-touch
foo.log
genius/spew.log
...
```

All right, that's the exact list of files that need renaming. Let's try generating some new names.

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname
> do
> echo mv $fname ${fname/.log/.LOG/}
> done
mv foo.log foo.LOG
mv genius/spew.log genius/spew.LOG
...
```

Yup, those are the commands we want to run to perform the renaming. So how do we do it for real? We could recall the command and edit out the **echo**, which would make **bash** execute the **mv** commands instead of just printing them. However, piping the commands to a separate instance of **bash** is less error-prone and requires less editing of the previous command.

When we type <Control-P>, we find that **bash** has thoughtfully collapsed our mini-script into a single line. To this condensed command line we simply add a pipe that sends the output to **bash -x**.

```
$ find . -type f -name '*.log' | grep -v .do-not-touch | while read fname; do
    echo mv $fname ${fname/.log/.LOG/}; done | bash -x
+ mv foo.log foo.LOG
+ mv genius/spew.log genius/spew.LOG
...
```

The **-x** option to **bash** prints each command before executing it.

We've now completed the actual renaming, but we'd still like to save this script so that we can use it again. **bash**'s built-in command **fc** is a lot like <Control-P>, but instead of returning the last command to the command line, it transfers the command to your editor of choice. Add a shebang line and usage comment, write the file to a plausible location (**~/bin** or **/usr/local/bin**, perhaps), make the file executable, and you have a script.

To summarize this approach:

- Develop the script (or script component) as a pipeline, one step at a time, entirely on the command line.
- Send output to standard output and check to be sure it looks right.
- At each step, use the shell's command history to recall pipelines and the shell's editing features to tweak them.
- Until the output looks right, you haven't actually done anything, so there's nothing to undo if the command is incorrect.
- Once the output is correct, execute the actual commands and verify that they worked as you intended.
- Use **fc** to capture your work, then clean it up and save it.

In the example above, we printed command lines and then piped them to a subshell for execution. This technique isn't universally applicable, but it's often helpful. Alternatively, you can capture output by redirecting it to a file. No matter what, wait until you see the right stuff in the preview before doing anything that's potentially destructive.

Input and output

The **echo** command is crude but easy. For more control over your output, use **printf**. It is a bit less convenient because you must explicitly put newlines where you want them (use "\n"), but it gives you the option to use tabs and enhanced number formatting in your the output. Compare the output from the following two commands.

```
$ echo "\taa\tbb\tcc\n"
\taa\tbb\tcc\n
$ printf "\taa\tbb\tcc\n"
    aa   bb   cc
```

Some systems have OS-level **echo** and **printf** commands, usually in **/bin** and **/usr/bin**, respectively. Although the commands and the shell built-ins are similar, they may diverge subtly in their specifics, especially in the case of **printf**. Either adhere to **bash**'s syntax or call the external **printf** with a full pathname.

You can use the **read** command to prompt for input. Here's an example:

```
#!/bin/bash

echo -n "Enter your name: "
read user_name

if [ -n "$user_name" ]; then
    echo "Hello $user_name!"
    exit 0
else
    echo "You did not tell me your name!"
    exit 1
fi
```

The **-n** in the **echo** command suppresses the usual newline, but you could also have used **printf** here. We cover the if statement's syntax shortly, but its effect should be obvious here. The -n in the if statement evaluates to true if its string argument is not null. Here's what the script looks like when run:

```
$ sh readexample
Enter your name: Ron
Hello Ron!
```

Command-line arguments and functions

Command-line arguments to a script become variables whose names are numbers. $1 is the first command-line argument, $2 is the second, and so on. $0 is the name by which the script was invoked. That could be something strange such as **../bin/example.sh**, so it's not a fixed value.

The variable $# contains the number of command-line arguments that were supplied, and the variable $* contains all the arguments at once. Neither of these variables includes or counts $0.

If you call a script without arguments or with inappropriate arguments, the script should print a short usage message to remind you how to use it. The example script below accepts two arguments, validates that the arguments are both directories, and displays them. If the arguments are invalid, the script prints a usage message and exits with a nonzero return code. If the caller of the script checks the return code, it will know that this script failed to execute correctly.

```
#!/bin/bash

function show_usage {
    echo "Usage: $0 source_dir dest_dir"
    exit 1
}

# Main program starts here

if [ $# -ne 2 ]; then
    show_usage
else # There are two arguments
    if [ -d $1 ]; then
        source_dir=$1
    else
        echo 'Invalid source directory'
        show_usage
    fi
    if [ -d $2 ]; then
        dest_dir=$2
    else
        echo 'Invalid destination directory'
        show_usage
    fi
fi

printf "Source directory is ${source_dir}\n"
printf "Destination directory is ${dest_dir}\n"
```

We created a separate show_usage function to print the usage message. If the script were later updated to accept additional arguments, the usage message would only have to be changed in one place.[5]

```
$ mkdir aaa bbb
$ sh showusage aaa bbb
Source directory is aaa
Destination directory is bbb
$ sh showusage foo bar
Invalid source directory
Usage: showusage source_dir dest_dir
```

5. Note that the error messages and usage message go to standard output. Shouldn't they go to standard error instead? That would in fact be more correct, but since this script isn't intended for use as a filter, the distinction is less important.

Arguments to **bash** functions are treated much like command-line arguments. The first argument becomes $1, and so on. As you can see in the example above, $0 remains the name of the script.

To make the previous example a bit more robust, we could make the show_usage routine accept an error code as an argument. That would allow a more definitive code to be returned for each different type of failure. The next code excerpt shows how that might look.

```
function show_usage {
    echo "Usage: $0 source_dir dest_dir"
    if [ $# -eq 0 ]; then
        exit 99 # Exit with arbitrary nonzero return code
    else
        exit $1
    fi
}
```

In this version of the routine, the argument is optional. Within a function, $# tells you how many arguments were passed in. The script exits with code 99 if no more-specific code is provided. But a specific value, for example,

```
show_usage 5
```

makes the script exit with that code after printing the usage message. (The shell variable $? contains the exit status of the last command executed, whether used inside a script or at the command line.)

The analogy between functions and commands is strong in **bash**. You can define useful functions in your **~/.bash_profile** file and then use them on the command line as if they were commands. For example, if your site has standardized on network port 7988 for the SSH protocol (a form of "security through obscurity"), you might define

```
function ssh {
    /usr/bin/ssh -p 7988 $*
}
```

in your **~/.bash_profile** to make sure **ssh** is always run with the option **-p 7988**.

Like many shells, bash has an **alias** mechanism that can reproduce this limited example even more concisely, but functions are more general and more powerful. Forget aliases and use functions.

Variable scope

Variables are global within a script, but functions can create their own local variables with a local declaration. Consider the following code.

```
#!/bin/bash

function localizer {
     echo "==> In function localizer, a starts as '$a'"
     local a
     echo "==> After local declaration, a is '$a'"
     a="localizer version"
     echo "==> Leaving localizer, a is '$a'"
}

a="test"
echo "Before calling localizer, a is '$a'"
localizer
echo "After calling localizer, a is '$a'"
```

The log below demonstrates that the local version of $a within the localizer function shadows the global variable $a. The global $a is visible within localizer until the local declaration is encountered; local is in fact a command that creates the local variable at the point when it's executed.

```
$ sh scopetest.sh
Before calling localizer, a is 'test'
==> In function localizer, a starts as 'test'
==> After local declaration, a is ''
==> Leaving localizer, a is 'localizer version'
After calling localizer, a is 'test'
```

Control flow

We've seen several if-then and if-then-else forms in this chapter already; they do exactly what you'd expect. The terminator for an if statement is fi. To chain your if clauses, you can use the elif keyword to mean "else if." For example:

```
if [ $base -eq 1 ] && [ $dm -eq 1 ]; then
     installDMBase
elif [ $base -ne 1 ] && [ $dm -eq 1 ]; then
     installBase
elif [ $base -eq 1 ] && [ $dm -ne 1 ]; then
     installDM
else
     echo '==> Installing nothing'
fi
```

Both the peculiar [] syntax for comparisons and the command-line optionlike names of the integer comparison operators (e.g., -eq) are inherited from the original Bourne shell's channeling of **/bin/test**. The brackets are actually a shorthand way of invoking **test** and are not a syntactic requirement of the if statement.[6]

6. In reality, these operations are now built into the shell and do not actually run **/bin/test**.

Table 2.2 shows the **bash** comparison operators for numbers and strings. **bash** uses textual operators for numbers and symbolic operators for strings, exactly the opposite of Perl.

Table 2.2 Elementary bash comparison operators

String	Numeric	True if
x = y	x -eq y	x is equal to y
x != y	x -ne y	x is not equal to y
x < y	x -lt y	x is less than y
x <= y	x -le y	x is less than or equal to y
x > y	x -gt y	x is greater than y
x >= y	x -ge y	x is greater than or equal to y
-n x	–	x is not null
-z x	–	x is null

bash shines in its options for evaluating the properties of files (again, courtesy of its **/bin/test** legacy). Table 2.3 shows a few of **bash**'s many file-testing and file-comparison operators.

Table 2.3 bash file evaluation operators

Operator	True if
-d *file*	*file* exists and is a directory
-e *file*	*file* exists
-f *file*	*file* exists and is a regular file
-r *file*	You have read permission on *file*
-s *file*	*file* exists and is not empty
-w *file*	You have write permission on *file*
file1 -nt *file2*	*file1* is newer than *file2*
file1 -ot *file2*	*file1* is older than *file2*

Although the elif form is useful, a case selection is often a better choice for clarity. Its syntax is shown below in a sample routine that centralizes logging for a script. Of particular note are the closing parenthesis after each condition and the two semicolons that follow the statement block to be executed when a condition is met. The case statement ends with esac.

```
# The log level is set in the global variable LOG_LEVEL. The choices
# are, from most to least severe, Error, Warning, Info, and Debug.

function logMsg {
    message_level=$1
    message_itself=$2
```

```
        if [ $message_level -le $LOG_LEVEL ]; then
            case $message_level in
                0) message_level_text="Error" ;;
                1) message_level_text="Warning" ;;
                2) message_level_text="Info" ;;
                3) message_level_text="Debug" ;;
                *) message_level_text="Other"
            esac
            echo "${message_level_text}: $message_itself"
        fi
    }
```

This routine illustrates the common "log level" paradigm used by many administrative applications. The code of the script generates messages at many different levels of detail, but only the ones that pass a globally set threshold, $LOG_LEVEL, are actually logged or acted upon. To clarify the importance of each message, the message text is preceded by a label that denotes its associated log level.

Loops

bash's for…in construct makes it easy to take some action for a group of values or files, especially when combined with filename globbing (the expansion of simple pattern-matching characters such as * and ? to form filenames or lists of filenames). The *.sh pattern in the for loop below returns a list of matching filenames in the current directory. The for statement then iterates through that list, assigning each filename in turn to the variable $file.

```
#!/bin/bash

suffix=BACKUP--`date +%Y%m%d-%H%M`

for script in *.sh; do
    newname="$script.$suffix"
    echo "Copying $script to $newname..."
    cp $script $newname
done
```

The output looks like this:

```
$ sh forexample
Copying rhel.sh to rhel.sh.BACKUP--20091210-1708...
Copying sles.sh to sles.sh.BACKUP--20091210-1708...
…
```

The filename expansion is not magic in this context; it works exactly as it does on the command line. Which is to say, the expansion happens first and the line is then processed by the interpreter in its expanded form.[7] You could just as well have entered the filenames statically, as in the line

```
for script in rhel.sh sles.sh; do
```

7. More accurately, the filename expansion is just a little bit magic in that it does maintain a notion of the atomicity of each filename. Filenames that contain spaces will go through the for loop in a single pass.

In fact, any whitespace-separated list of things, including the contents of a variable, works as a target of for…in.

bash also has the more familiar for loop from traditional programming languages in which you specify starting, increment, and termination clauses. For example:

```
for (( i=0 ; i < $CPU_COUNT ; i++ )); do
    CPU_LIST="$CPU_LIST $i"
done
```

The next example illustrates **bash**'s while loop, which is useful for processing command-line arguments and for reading the lines of a file.

```
#!/bin/bash

exec 0<$1
counter=1
while read line; do
    echo "$counter: $line"
    $((counter++))
done
```

Here's what the output looks like:

```
ubuntu$ sh whileexample /etc/passwd
1: root:x:0:0:Superuser:/root:/bin/bash
2: bin:x:1:1:bin:/bin:/bin/bash
3: daemon:x:2:2:Daemon:/sbin:/bin/bash
…
```

This scriptlet has a couple of interesting features. The exec statement redefines the script's standard input to come from whatever file is named by the first command-line argument.[8] The file must exist or the script generates an error.

The read statement within the while clause is in fact a shell built-in, but it acts like an external command. You can put external commands in a while clause as well; in that form, the while loop terminates when the external command returns a nonzero exit status.

The $((counter++)) expression is an odd duck, indeed. The $((…)) notation forces numeric evaluation. It also makes optional the use of $ to mark variable names. The ++ is the familiar postincrement operator from C and other languages. It returns the value of the variable to which it's attached, but has the side effect of incrementing that variable's value as well.

The $((…)) shenanigans work in the context of double quotes, so the body of the loop could be collapsed down to one line.

8. Depending on the invocation, exec can also have the more familiar meaning "stop this script and transfer control to another script or expression." It's yet another shell oddity that both functions are accessed through the same statement.

```
while read line; do
    echo "$((counter++)): $line"
done
```

Arrays and arithmetic

Sophisticated data structures and calculations aren't **bash**'s forte. But it does at least offer arrays and arithmetic.

All **bash** variables are string valued, so **bash** does not distinguish between the number 1 and the character string "1" in assignments. The difference lies in how the variables are used. The following code illustrates the distinction:

```
#!/bin/bash

a=1
b=$((2))

c=$a+$b
d=$(($a+$b))

echo "$a + $b = $c \t(plus sign as string literal)"
echo "$a + $b = $d \t(plus sign as arithmetic addition)"
```

This script produces the output

```
1 + 2 = 1+2  (plus sign as string literal)
1 + 2 = 3    (plus sign as arithmetic addition)
```

Note that the plus sign in the assignment to $c does not even act as a concatenation operator for strings. It's just a literal character. That line is equivalent to

```
c="$a+$b"
```

To force numeric evaluation, you enclose an expression in $((...)), as shown with the assignment to $d above. But even this precaution does not result in $d receiving a numeric value; the value is still stored as the string "3".

bash has the usual assortment of arithmetic, logical, and relational operators; see the man page for details.

Arrays in **bash** are a bit strange, and they're not often used. Nevertheless, they're available if you need them. Literal arrays are delimited by parentheses, and the elements are separated by whitespace. You can use quoting to include literal spaces in an element.

```
example=(aa 'bb cc' dd)
```

Use ${*array_name*[*subscript*]} to access individual elements. Subscripting begins at zero. The subscripts * and @ refer to the array as a whole, and the special forms ${#*array_name*[*]} and ${#*array_name*[@]} yield the number of elements in the array. Don't misremember these as the more logical-seeming ${#*array_name*}; that is in fact the length of the array's first element (equivalent to ${#*array_name*[0]}).

You might think that $example[1] would be an unambiguous reference to the second element of the array, but **bash** parses this string as $example (a shorthand reference to $example[0]) plus the literal string [1]. Always include the curly braces when referring to array variables—no exceptions.

Here's a quick script that illustrates some of the features and pitfalls of array management in **bash**:

```
#!/bin/bash

example=(aa 'bb cc' dd)
example[3]=ee

echo "example[@] = ${example[@]}"
echo "example array contains ${#example[@]} elements"

for elt in "${example[@]}"; do
    echo "  Element = $elt"
done
```

Its output is

```
$ sh arrays
example[@] = aa bb cc dd ee
example array contains 4 elements
    Element = aa
    Element = bb cc
    Element = dd
    Element = ee
```

This example seems straightforward, but only because we've constructed it to be well behaved. Pitfalls await the unwary. For example, replacing the for line with

```
for elt in ${example[@]}; do
```

(without quotes around the array expression) also works fine, but instead of four array elements it yields five: aa, bb, cc, dd, and ee.

The underlying issue is that all **bash** variables are still essentially strings, so the illusion of arrays is wobbly at best. Subtleties regarding when and how strings are separated into elements abound. You can use Perl or Python, or google for Mendel Cooper's *Advanced Bash-Scripting Guide* to investigate the nuances.

2.3 REGULAR EXPRESSIONS

Regular expressions are supported by most modern languages, though some take them more to heart than others. They're also used by UNIX commands such as **grep** and **vi**. They are so common that the name is usually shortened to "regex." Entire books have been written about how to harness their power, and they have been the subject of numerous doctoral dissertations.

The filename matching and expansion performed by the shell when it interprets command lines such as **wc -l *.pl** *is not* a form of regex matching. It's a different system called "shell globbing," and it uses a different and simpler syntax.

Regular expressions are powerful, but they cannot recognize all possible grammars. Their most notable weakness is that they cannot recognize nested delimiters. For example, it's not possible to write a regular expression that recognizes valid arithmetic expressions when parentheses are allowed for grouping.

Regular expressions reached the apex of their power and perfection in Perl. In fact, Perl's pattern matching features are so elaborate that it's not really accurate to call them an implementation of regular expressions. Perl patterns can match nested delimiters, recognize palindromes, and match an arbitrary string of As followed by the same number of Bs—all feats beyond the reach of regular expressions. However, Perl can process vanilla regular expressions as well.

Perl's pattern matching language remains the industry benchmark, and it has been widely adopted by other languages and tools. Philip Hazel's PCRE (Perl-compatible regular expression) library makes it relatively easy for developers to incorporate the language into their own projects.

Regular expressions are not themselves a scripting language, but they're so useful that they merit featured coverage in any discussion of scripting; hence, this section.[9] Here, we discuss them in their basic form with a few of Perl's refinements.

The matching process

Code that evaluates a regular expression attempts to match a single given text string to a single given pattern. The "text string" to match can be very long and can contain embedded newlines. It's often convenient to use a regex to match the contents of an entire file or HTML document.

For the matcher to declare success, the entire search pattern must match a contiguous section of the search text. However, the pattern can match at any position. After a successful match, the evaluator returns the text of the match along with a list of matches for any specially delimited subsections of the pattern.

Literal characters

In general, characters in a regular expression match themselves. So the pattern

```
I am the walrus
```

matches the string "I am the walrus" and that string only. Since it can match anywhere in the search text, the pattern can be successfully matched to the string "I am the egg man. I am the walrus. Koo koo ka-choo!" However, the actual match is limited to the "I am the walrus" portion. Matching is case sensitive.

9. Perl guru Tom Christiansen commented, "I don't know what a 'scripting language' is, but I agree that regular expressions are neither procedural nor functional languages. Rather, they are a logic-based or declarative language, a class of languages that also includes Prolog and Makefiles. And BNFs. One might also call them rule-based languages. I prefer to call them declarative languages myself."

Special characters

Table 2.4 shows the meanings of some common special symbols that can appear in regular expressions. These are just the basics—there are many, many more.

Table 2.4 Special characters in regular expressions (common ones)

Symbol	What it matches or does
.	Matches any character
[*chars*]	Matches any character from a given set
[^*chars*]	Matches any character not in a given set
^	Matches the beginning of a line
$	Matches the end of a line
\w	Matches any "word" character (same as [A-Za-z0-9_])
\s	Matches any whitespace character (same as [\f\t\n\r])[a]
\d	Matches any digit (same as [0-9])
|	Matches either the element to its left or the one to its right
(*expr*)	Limits scope, groups elements, allows matches to be captured
?	Allows zero or one match of the preceding element
*	Allows zero, one, or many matches of the preceding element
+	Allows one or more matches of the preceding element
{*n*}	Matches exactly *n* instances of the preceding element
{*min,*}	Matches at least *min* instances (note the comma)
{*min,max*}	Matches any number of instances from *min* to *max*

a. That is, a space, a form feed, a tab, a newline, or a return

Many special constructs, such as + and |, affect the matching of the "thing" to their left or right. In general, a "thing" is a single character, a subpattern enclosed in parentheses, or a character class enclosed in square brackets. For the | character, however, thingness extends indefinitely to both left and right. If you want to limit the scope of the vertical bar, enclose the bar and both things in their own set of parentheses. For example,

```
I am the (walrus|egg man)\.
```

matches either "I am the walrus." or "I am the egg man.". This example also demonstrates escaping of special characters (here, the dot). The pattern

```
(I am the (walrus|egg man)\. ?){1,2}
```

matches any of the following:

- I am the walrus.
- I am the egg man.
- I am the walrus. I am the egg man.
- I am the egg man. I am the walrus.

Unfortunately, it also matches "I am the egg man. I am the egg man.". (What kind of sense does that make?) More importantly, it also matches "I am the walrus. I am the egg man. I am the walrus.", even though the number of repetitions is explicitly capped at two. That's because the pattern need not match the entire search text. Here, the regex matches two sentences and terminates, declaring success. It simply doesn't care that another repetition is available.

It is a common error to confuse the regular expression metacharacter * (the zero-or-more quantifier) with the shell's * globbing character. The regex version of the star needs something to modify; otherwise, it won't do what you expect. Use .* if any sequence of characters (including no characters at all) is an acceptable match.

Example regular expressions

In the United States, postal ("zip") codes have either five digits or five digits followed by a dash and four more digits. To match a regular zip code, you must match a five-digit number. The following regular expression fits the bill:

```
^\d{5}$
```

The ^ and $ match the beginning and end of the search text but do not actually correspond to characters in the text; they are "zero-width assertions." These characters ensure that only texts consisting of exactly five digits match the regular expression—the regex will not match five digits within a larger string. The `\d` escape matches a digit, and the quantifier `{5}` says that there must be exactly five digit matches.

To accommodate either a five-digit zip code or an extended zip+4, add an optional dash and four additional digits:

```
^\d{5}(-\d{4})?$
```

The parentheses group the dash and extra digits together so that they are considered one optional unit. For example, the regex won't match a five-digit zip code followed by a dash. If the dash is present, the four-digit extension must be present as well or there is no match.

A classic demonstration of regex matching is the following expression,

```
M[ou]'?am+[ae]r ([AEae]l[- ])?[GKQ]h?[aeu]+([dtz][dhz]?)+af[iy]
```

which matches most of the variant spellings of the name of Libyan head of state Moammar Gadhafi, including

• Muammar al-Kaddafi	(BBC)
• Moammar Gadhafi	(Associated Press)
• Muammar al-Qadhafi	(Al-Jazeera)
• Mu'ammar Al-Qadhafi	(U.S. Department of State)

Do you see how each of these would match the pattern?

This regular expression also illustrates how quickly the limits of legibility can be reached. Many regex systems (including Perl's) support an x option that ignores literal whitespace in the pattern and enables comments, allowing the pattern to be spaced out and split over multiple lines. You can then use whitespace to separate logical groups and clarify relationships, just as you would in a procedural language. For example:

```
M [ou] '? a m+ [ae] r    # First name: Mu'ammar, Moamar, etc.
\s                       # Whitespace; can't use a literal space here
(                        # Group for optional last name prefix
  [AEae] l               #    Al, El, al, or el
  [-\s]                  #    Followed by dash or space
)?
[GKQ] h? [aeu]+          # Initial syllable of last name: Kha, Qua, etc.
(                        # Group for consonants at start of 2nd syllable
  [dtz] [dhz]?           #    dd, dh, etc.
)+
af [iy]
```

This helps a little bit, but it's still pretty easy to torture later readers of your code. So be kind: if you can, use hierarchical matching and multiple small matches instead of trying to cover every possible situation in one large regular expression.

Captures

When a match succeeds, every set of parentheses becomes a "capture group" that records the actual text that it matched. The exact manner in which these pieces are made available to you depends on the implementation and context. In Perl, you can access the results as a list or as a sequence of numbered variables.

Since parentheses can nest, how do you know which match is which? Easy—the matches arrive in the same order as the opening parentheses. There are as many captures as there are opening parentheses, regardless of the role (or lack of role) that each parenthesized group played in the actual matching. When a parenthesized group is not used (e.g., Mu(')?ammar when matched against "Muammar"), its corresponding capture is empty.

If a group is matched more than once, only the contents of the last match are returned. For example, with the pattern

```
(I am the (walrus|egg man)\. ?){1,2}
```

matching the text

```
I am the egg man. I am the walrus.
```

there are two results, one for each set of parentheses:

```
I am the walrus.
walrus
```

Note that both capture groups actually matched twice. However, only the last text to match each set of parentheses is actually captured.

Greediness, laziness, and catastrophic backtracking

Regular expressions match from left to right. Each component of the pattern matches the longest possible string before yielding to the next component, a characteristic known as greediness.

If the regex evaluator reaches a state from which a match cannot be completed, it unwinds a bit of the candidate match and makes one of the greedy atoms give up some of its text. For example, consider the regex `a*aa` being matched against the input text "aaaaaa".

At first, the regex evaluator assigns the entire input to the `a*` portion of the regex, because the `a*` is greedy. When there are no more a's to match, the evaluator goes on to try to match the next part of the regex. But oops, it's an `a`, and there is no more input text that can match an `a`; time to backtrack. The `a*` has to give up one of the a's it has matched.

Now the evaluator can match `a*a`, but it still cannot match the last `a` in the pattern. So it backtracks again and takes away a second a from the `a*`. Now the second and third `a`'s in the pattern both have a's to pair with, and the match is complete.

This simple example illustrates some important general points. First, greedy matching plus backtracking makes it expensive to match apparently simple patterns such as <img.*></tr> when processing entire files.[10] The .* portion starts by matching everything from the first <img to the end of the input, and only through repeated backtracking does it contract to fit the local tags.

Furthermore, the ></tr> that this pattern binds to is the *last possible* valid match in the input, which is probably not what you want. More likely, you meant to match an <img> followed by a </tr> tag. A better way to write this pattern is <img[^>]*></tr>, which allows the initial wild-card match to expand only to the end of the current tag because it cannot cross a right-angle-bracket boundary.

You can also use lazy (as opposed to greedy) wild card operators: *? instead of *, and +? instead of +. These versions match as few characters of the input as they can. If that fails, they match more. In many situations, these operators are more efficient and closer to what you want than the greedy versions.

Note, however, that they can produce different matches than the greedy operators; the difference is more than just one of implementation. In our HTML example, the lazy pattern would be <img.*?></tr>. But even here, the .*? could eventually

10. Although this section shows HTML excerpts as examples of text to be matched, regular expressions are not really the right tool for this job. Our external reviewers were uniformly aghast. Perl and Python both have excellent add-ons that parse HTML documents the proper way. You can then access the portions you're interested in with XPath selectors. See the Wikipedia page for XPath and the respective languages' module repositories for details.

grow to include unwanted >'s because the next tag after an <img> might not be a </tr>. Again, probably not what you want.

Patterns with multiple wild-card sections can cause exponential behavior in the regex evaluator, especially if portions of the text can match several of the wild-card expressions and especially if the search text does not in fact match the pattern. This situation is not as unusual as it might sound, especially when pattern matching with HTML. Very often, you'll want to match certain tags followed by other tags, possibly separated by even more tags, a recipe that may require the regex evaluator to try many possible combinations.

Regex guru Jan Goyvaerts calls this phenomenon "catastrophic backtracking" and writes about it in his blog; see regular-expressions.info/catastrophic.html for details and some good solutions.

A couple of take-home points from all this:

- If you can do pattern matching line-by-line rather than file-at-a-time, there is much less risk of poor performance.
- Even though regex notation makes greedy operators the default, they probably shouldn't be. Use lazy operators.
- All instances of .* are inherently suspicious and should be scrutinized.

2.4 Perl programming

Perl, created by Larry Wall, was the first of the truly great scripting languages. It offers vastly more power than **bash**, and well-written Perl code is quite easy to read. On the other hand, Perl does not impose much stylistic discipline on developers, so Perl code written without regard for readability can be cryptic. Perl has been accused of being a write-only language.

Here we describe Perl 5, the version that has been standard for the last decade. Perl 6 is a major revision that's still in development. See perl6.org for details.

Either Perl or Python (discussed starting on page 66) is a better choice for system administration work than traditional programming languages such as C, C++, C#, and Java. They can do more, in fewer lines of code, with less painful debugging, and without the hassle of compilation.

Language choice usually comes down to personal preference or to standards forced upon you by an employer. Both Perl and Python offer libraries of community-written modules and language extensions. Perl has been around longer, so its offerings extend further into the long tail of possibilities. For common system administration tasks, however, the support libraries are roughly equivalent.

Perl's catch phrase is that "there's more than one way to do it." So keep in mind that there are other ways of doing most of what you read in this section.

Perl statements are separated by semicolons.[11] Comments start with a hash mark (#) and continue to the end of the line. Blocks of statements are enclosed in curly braces. Here's a simple "hello, world!" program:

```
#!/usr/bin/perl
print "Hello, world!\n";
```

As with **bash** programs, you must either **chmod +x** the executable file or invoke the Perl interpreter directly.

```
$ chmod +x helloworld
$ ./helloworld
Hello, world!
```

Lines in a Perl script are not shell commands; they're Perl code. Unlike **bash**, which lets you assemble a series of commands and call it a script, Perl does not look outside itself unless you tell it to. That said, Perl provides many of the same conventions as **bash**, such as the use of back-ticks to capture the output from a command.

Variables and arrays

Perl has three fundamental data types: scalars (that is, unitary values such as numbers and strings), arrays, and hashes. Hashes are also known as associative arrays. The type of a variable is always obvious because it's built into the variable name: scalar variables start with $, array variables start with @, and hash variables start with %.

In Perl, the terms "list" and "array" are often used interchangeably, but it's perhaps more accurate to say that a list is a series of values and an array is a variable that can hold such a list. The individual elements of an array are scalars, so like ordinary scalar variables, their names begin with $. Array subscripting begins at zero, and the index of the highest element in array `@a` is `$#a`. Add 1 to that to get the array's size.

The array `@ARGV` contains the script's command-line arguments. You can refer to it just like any other array.

The following script demonstrates the use of arrays:

```
#!/usr/bin/perl

@items = ("socks", "shoes", "shorts");
printf "There are %d articles of clothing.\n", $#items + 1;
print "Put on ${items[2]} first, then ", join(" and ", @items[0,1]), ".\n";
```

The output:

```
$ perl clothes
There are 3 articles of clothing.
Put on shorts first, then socks and shoes.
```

11. Since semicolons are separators and not terminators, the last one in a block is optional.

There's a lot to see in just these few lines. At the risk of blurring our laser-like focus, we include several common idioms in each of our Perl examples. We explain the tricky parts in the text following each example. If you read the examples carefully (don't be a wimp, they're short!), you'll have a working knowledge of the most common Perl forms by the end of this chapter.

Array and string literals

In this example, notice first that (…) creates a literal list. Individual elements of the list are strings, and they're separated by commas. Once the list has been created, it is assigned to the variable @items.

Perl does not strictly require that all strings be quoted. In this particular case, the initial assignment of @items works just as well without the quotes.

```
@items = (socks, shoes, shorts);
```

Perl calls these unquoted strings "barewords," and they're an interpretation of last resort. If something doesn't make sense in any other way, Perl tries to interpret it as a string. In a few limited circumstances, this makes sense and keeps the code clean. However, this is probably not one of those cases. Even if you prefer to quote strings consistently, be prepared to decode other people's quoteless code.

The more Perly way to initialize this array is with the qw (quote words) operator. It is in fact a form of string quotation, and like most quoted entities in Perl, you can choose your own delimiters. The form

```
@items = qw(socks shoes shorts);
```

is the most traditional, but it's a bit misleading since the part after the qw is no longer a list. It is in fact a string to be split at whitespace to form a list. The version

```
@items = qw[socks shoes shorts];
```

works, too, and is perhaps a bit truer to the spirit of what's going on. Note that the commas are gone since their function has been subsumed by qw.

Function calls

Both **print** and **printf** accept an arbitrary number of arguments, and the arguments are separated by commas. But then there's that join(…) thing that looks like some kind of function call; how is it different from **print** and **printf**?

In fact, it's not; **print**, **printf**, and **join** are all plain-vanilla functions. Perl allows you to omit the parentheses in function calls when this does not cause ambiguity, so both forms are common. In the **print** line above, the parenthesized form distinguishes the arguments to **join** from those that go to **print**.

We can tell that the expression @items[0,1] must evaluate to some kind of list since it starts with @. This is in fact an "array slice" or subarray, and the 0,1 subscript lists the indexes of the elements to be included in the slice. Perl accepts a range of values here, too, as in the equivalent expression @items[0..1]. A single numeric subscript would be acceptable here as well: @items[0] is a list containing one scalar, the string “socks”. In this case, it’s equivalent to the literal ("socks").

Arrays are automatically expanded in function calls, so in the expression

```
join(" and ", @items[0,1])
```

join receives three string arguments: “ and ”, “socks”, and “shoes”. It concatenates its second and subsequent arguments, inserting a copy of the first argument between each pair. The result is “socks and shoes”.

Type conversions in expressions

In the **printf** line, $#items + 1 evaluates to the number 3. As it happens, $#items is a numeric value, but that’s not why the expression is evaluated arithmetically; "2" + 1 works just as well. The magic is in the + operator, which always implies arithmetic. It converts its arguments to numbers and produces a numeric result. Similarly, the dot operator (.), which concatenates strings, converts its operands as needed: "2" . (12 ** 2) yields “2144”.

String expansions and disambiguation of variable references

As in **bash**, double-quoted strings are subject to variable expansion. Also as in **bash**, you can surround variable names with curly braces to disambiguate them if necessary, as with ${items[2]}. (Here, the braces are used only for illustration; they are not needed.) The $ clues you in that the expression is going to evaluate to a scalar. @items is the array, but any individual element is itself a scalar, and the naming conventions reflect this fact.

Hashes

A hash (also known as an associative array) represents a set of key/value pairs. You can think of a hash as an array whose subscripts (keys) are arbitrary scalar values; they do not have to be numbers. But in practice, numbers and strings are the usual keys.

Hash variables have % as their first character (e.g., %myhash), but as in the case of arrays, individual values are scalar and so begin with a $. Subscripting is indicated with curly braces rather than square brackets, e.g., $myhash{'ron'}.

Hashes are an important tool for system administrators. Nearly every script you write will use them. In the code below, we read in the contents of a file, parse it according to the rules for **/etc/passwd**, and build a hash of the entries called %names_by_uid. The value of each entry in the hash is the username associated with that UID.

```
#!/usr/bin/perl

while ($_ = <>) {
      ($name, $pw, $uid, $gid, $gecos, $path, $sh) = split /:/;
      $names_by_uid{$uid} = $name;
}
```

```
%uids_by_name = reverse %names_by_uid;

print "\$names_by_uid{0} is $names_by_uid{0}\n";
print "\$uids_by_name{'root'} is $uids_by_name{'root'}\n";
```

As in the previous script example, we've packed a couple of new ideas into these lines. Before we go over each of these nuances, here's the output of the script:

```
$ perl hashexample /etc/passwd
$names_by_uid{0} is root
$uids_by_name{'root'} is 0
```

The while ($_ = <>) reads input one line at a time and assigns it to the variable named $_; the value of the entire assignment statement is the value of the right-hand side, just as in C. When you reach the end of the input, the <> returns a false value and the loop terminates.

To interpret <>, Perl checks the command line to see if you named any files there. If you did, it opens each file in sequence and runs the file's contents through the loop. If you didn't name any files on the command line, Perl takes the input to the loop from standard input.

Within the loop, a series of variables receive the values returned by split, a function that chops up its input string by using the regular expression passed to it as the field separator. Here, the regex is delimited by slashes; this is just another form of quoting, one that's specialized for regular expressions but similar to the interpretation of double quotes. We could just as easily have written split ':' or split ":".

The string that split is to divide at colons is never explicitly specified. When split's second argument is missing, Perl assumes you want to split the value of $_. Clean! Truth be told, even the pattern is optional; the default is to split at whitespace but ignore any leading whitespace.

But wait, there's more. Even the original *assignment* of $_, back at the top of the loop, is unnecessary. If you simply say

```
while (<>) {
```

Perl automatically stores each line in $_. You can process lines without ever making an explicit reference to the variable in which they're stored. Using $_ as a default operand is common, and Perl allows it more or less wherever it makes sense.

In the multiple assignment that captures the contents of each **passwd** field,

```
    ($name, $pw, $uid, $gid, $gecos, $path, $sh) = split /:/;
```

the presence of a list on the left hand side creates a "list context" for split that tells it to return a list of all fields as its result. If the assignment were to a scalar variable, for example,

```
$n_fields = split /:/;
```

split would run in "scalar context" and return only the number of fields that it found. Functions you write can distinguish between scalar and list contexts, too, by using the wantarray function. It returns a true value in list context, a false value in scalar context, and an undefined value in void context.

The line

```
%uids_by_name = reverse %names_by_uid;
```

has some hidden depths, too. A hash in list context (here, as an argument to the reverse function) evaluates to a list of the form (key1, value1, key2, value2, …). The reverse function reverses the order of the list, yielding (valueN, keyN, …, value1, key1). Finally, the assignment to the hash variable %uids_by_name converts this list as if it were (key1, value1, …), thereby producing a permuted index.

References and autovivification

These are advanced topics, but we'd be remiss if we didn't at least mention them. Here's the executive summary. Arrays and hashes can only hold scalar values, but you will often want to store other arrays and hashes within them. For example, returning to our previous example of parsing the **/etc/passwd** file, you might want to store *all* the fields of each **passwd** line in a hash indexed by UID.

You can't store arrays and hashes, but you can store *references* (that is, pointers) to arrays and hashes, which are themselves scalars. To create a reference to an array or hash, you precede the variable name with a backslash (e.g., \@array) or use reference-to-array or reference-to-hash literal syntax. For example, our passwd-parsing loop would become something like this:

```
while (<>) {
    $array_ref = [ split /:/ ];
    $passwd_by_uid{$array_ref->[2]} = $array_ref;
}
```

The square brackets return a reference to an array containing the results of the split. The notation $array_ref->[2] refers to the UID field, the third member of the array referenced by $array_ref.

$array_ref[2] won't work here because we haven't defined an @array_ref array; $array_ref and @array_ref are different variables. Furthermore, you won't receive an error message if you mistakenly use $array_ref[2] here because @array_ref is a perfectly legitimate name for an array; you just haven't assigned it any values.

This lack of warnings may seem like a problem, but it's arguably one of Perl's nicest features, a feature known as "autovivification." Because variable names and referencing syntax always make clear the structure of the data you are trying to access, you need never create any intermediate data structures by hand. Simply make an assignment at the lowest possible level, and the intervening structures materialize automatically. For example, you can create a hash of references to arrays whose contents are references to hashes with a single assignment.

Regular expressions in Perl

You use regular expressions in Perl by "binding" strings to regex operations with the =~ operator. For example, the line

```
if ($text =~ m/ab+c/) {
```

checks to see whether the string stored in $text matches the regular expression ab+c. To operate on the default string, $_, you can simply omit the variable name and binding operator. In fact, you can omit the m, too, since the operation defaults to matching:

```
if (/ab+c/) {
```

Substitutions work similarly:

```
$text =~ s/etc\./and so on/g;        # Substitute text in $text, OR
s/etc\./and so on/g;                 # Apply to $_
```

We sneaked in a g option to replace all instances of "etc." with "and so on", rather than just replacing the first instance. Other common options are i to ignore case, s to make dot (.) match newlines, and m to make the ^ and $ tokens match at the beginning and end of individual lines rather than only at the beginning and end of the search text.

A couple of additional points are illustrated in the following script:

```
#!/usr/bin/perl

$names = "huey dewey louie";
$regex = '(\w+)\s+(\w+)\s+(\w+)';

if ($names =~ m/$regex/) {
    print "1st name is $1.\n2nd name is $2.\n3rd name is $3.\n";
    $names =~ s/$regex/\2 \1/;
    print "New names are \"${names}\".\n";
} else {
    print qq{"$names" did not match "$regex".\n};
}
```

The output:

```
$ perl testregex
1st name is huey.
2nd name is dewey.
3rd name is louie.
New names are "dewey huey".
```

This example shows that variables expand in // quoting, so the regular expression need not be a fixed string. qq is another name for the double-quote operator.

After a match or substitution, the contents of the variables $1, $2, and so on correspond to the text matched by the contents of the capturing parentheses in the regular expression. The contents of these variables are also available during the replacement itself, in which context they are referred to as \1, \2, etc.

Input and output

When you open a file for reading or writing, you define a "filehandle" to identify the channel. In the example below, INFILE is the filehandle for **/etc/passwd** and OUTFILE is the filehandle associated with **/tmp/passwd**. The while loop condition is <INFILE>, which is similar to the <> we have seen before but specific to a particular filehandle. It reads lines from the filehandle INFILE until the end of file, at which time the while loop ends. Each line is placed in the variable $_.

```
#!/usr/bin/perl

open(INFILE, "</etc/passwd") or die "Couldn't open /etc/passwd";
open(OUTFILE, ">/tmp/passwd") or die "Couldn't open /tmp/passwd";

while (<INFILE>) {
    ($name, $pw, $uid, $gid, $gecos, $path, $sh) = split /:/;
    print OUTFILE "$uid\t$name\n";
}
```

open returns a true value if the file is successfully opened, short-circuiting (rendering unnecessary) the evaluation of the die clauses. Perl's or operator is similar to || (which Perl also has), but at lower precedence. or is a generally a better choice when you want to emphasize that everything on the left will be fully evaluated before Perl turns its attention to the consequences of failure.

Perl's syntax for specifying how you want to use each file (read? write? append?) mirrors that of the shell. You can also use "filenames" such as "/bin/df|" to open pipes to and from shell commands.

Control flow

The example below is a Perl version of our earlier **bash** script that validated its command-line arguments. You might want to refer to the **bash** version on page 41 for comparison. Note that Perl's if construct has no then keyword or terminating word, just a block of statements enclosed in curly braces.

You can also add a postfix if clause (or its negated version, unless) to an individual statement to make that statement's execution conditional.

```
#!/usr/bin/perl

sub show_usage {
    print shift, "\n" if scalar(@_);
    print "Usage: $0 source_dir dest_dir\n";
    exit scalar(@_) ? shift : 1;
}
if (@ARGV != 2) {
    show_usage;
} else { # There are two arguments
    ($source_dir, $dest_dir) = @ARGV;
    show_usage "Invalid source directory" unless -d $source_dir;
    -d $dest_dir or show_usage "Invalid destination directory";
}
```

Here, the two lines that use Perl's unary -d operator to validate the directory-ness of $source_dir and $dest_dir are equivalent. The second form (with -d at the start of the line) has the advantage of putting the actual assertion at the beginning of the line, where it's most noticeable. However, the use of or to mean "otherwise" is a bit tortured; some readers of the code may find it confusing.

Evaluating an array variable in scalar context (specified by the scalar operator in this example) returns the number of elements in the array. This is 1 more than the value of $#array; as always in Perl, there's more than one way to do it.

Perl functions receive their arguments in the array named @_. It's common practice to access them with the shift operator, which removes the first element of the argument array and returns its value.

This version of the show_usage function accepts an optional error message to be printed. If you provide an error message, you can also provide a specific exit code. The trinary ?: operator evaluates its first argument; if the result is true, the result of the entire expression is the second argument; otherwise, the third.

As in **bash**, Perl has a dedicated "else if" condition, but its keyword is elsif rather than elif. (For you who use both languages, these fun, minute differences either keep you mentally nimble or drive you insane.)

As Table 2.5 shows, Perl's comparison operators are the opposite of **bash**'s; strings use textual operators, and numbers use traditional algebraic notation. Compare with Table 2.2 on page 44.

Table 2.5 Elementary Perl comparison operators

String	Numeric	True if
x eq y	x = y	x is equal to y
x ne y	x != y	x is not equal to y
x lt y	x < y	x is less than y
x le y	x <= y	x is less than or equal to y
x gt y	x > y	x is greater than y
x ge y	x >= y	x is greater than or equal to y

In Perl, you get all the file-testing operators shown in Table 2.3 on page 44 except for the -nt and -ot operators, which are available in **bash** only.

Like **bash**, Perl has two types of for loops. The more common form iterates through an explicit list of arguments. For example, the code below iterates through a list of animals, printing one per line.

```
@animals = qw(lions tigers bears);
foreach $animal (@animals) {
    print "$animal \n" ;
}
```

The more traditional C-style for loop is also available:

```
for ($counter=1; $counter <= 10; $counter++) {
    printf "$counter ";
}
```

We've shown these with the traditional for and foreach labels, but those are in fact the same keyword in Perl and you can use whichever form you prefer.

Versions of Perl before 5.10 (2007) have no explicit case or switch statement, but there are several ways to accomplish the same thing. In addition to the obvious-but-clunky option of cascading if statements, another possibility is to use a for statement to set the value of $_ and provide a context from which last can escape:

```
for ($ARGV[0]) {

    m/^websphere/   && do { print "Install for websphere\n"; last; };
    m/^tomcat/      && do { print "Install for tomcat\n" ; last; };
    m/^geronimo/    && do { print "Install for geronimo\n"; last; };

    print "Invalid option supplied.\n"; exit 1;
}
```

The regular expressions are compared with the argument stored in $_. Unsuccessful matches short-circuit the && and fall through to the next test case. Once a regex matches, its corresponding do block is executed. The last statements escape from the for block immediately.

Accepting and validating input

The script below combines many of the Perl constructs we've reviewed over the last few pages, including a subroutine, some postfix if statements, and a for loop. The program itself is merely a wrapper around the main function get_string, a generic input validation routine. This routine prompts for a string, removes any trailing newline, and verifies that the string is not null. Null strings cause the prompt to be repeated up to three times, after which the script gives up.

```
#!/usr/bin/perl

$maxatt = 3; # Maximum tries to supply valid input

sub get_string {
    my ($prompt, $response) = shift;
    # Try to read input up to $maxatt times
    for (my $attempts = 0; $attempts < $maxatt; $attempts++) {
        print "Please try again.\n" if $attempts;
        print "$prompt: ";
        $response = readline(*STDIN);
        chomp($response);
        return $response if $response;
    }
    die "Too many failed input attempts";
}
```

```
# Get names with get_string and convert to uppercase
$fname = uc get_string "First name";
$lname = uc get_string "Last name";
printf "Whole name: $fname $lname\n";
```

The output:

```
$ perl validate
First name: John Ball
Last name: Park
Whole name: JOHN BALL PARK
```

The get_string function and the for loop both illustrate the use of the my operator to create variables of local scope. By default, all variables are global in Perl.

The list of local variables for get_string is initialized with a single scalar drawn from the routine's argument array. Variables in the initialization list that have no corresponding value (here, $response) remain undefined.

The *STDIN passed to the readline function is a "typeglob," a festering wart of language design. It's best not to inquire too deeply into what it really means, lest one's head explode. The short explanation is that Perl filehandles are not first-class data types, so you must generally put a star in front of their names to pass them as arguments to functions.

In the assignments for $fname and $lname, the uc (convert to uppercase) and get_string functions are both called without parentheses. Since there is no possibility of ambiguity given the single argument, this works fine.

Perl as a filter

You can use Perl without a script by putting isolated expressions on the command line. This is a great way to do quick text transformations and one that largely obsoletes older filter programs such as **sed**, **awk**, and **tr**.

Use the **-pe** command-line option to loop through STDIN, run a simple expression on each line, and print the result. For example, the command

```
ubuntu$ perl -pe 's#/bin/sh$#/bin/bash#' /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/bash
…
```

replaces **/bin/sh** at the end of lines in **/etc/passwd** with **/bin/bash**, emitting the transformed **passwd** file to STDOUT. You may be more accustomed to seeing the text substitution operator with slashes as delimiters (e.g., **s/foo/bar/**), but Perl allows any character. Here, the search text and replacement text both contain slashes, so it's simpler to use # as the delimiter. If you use paired delimiters, you must use four of them instead of the normal three, e.g., **s(foo)(bar)**.

Perl's **-a** option turns on autosplit mode, which separates input lines into fields that are stored in the array named @F. Whitespace is the default field separator, but you can set another separator pattern with the **-F** option.

Autosplit is handy to use in conjunction with **-p** or its nonautoprinting variant, **-n**. For example, the commands below use **perl -ane** to slice and dice the output from two variations of **df**. The third line then runs **join** to combine the two sets of fields on the Filesystem field, producing a composite table that includes fields drawn from both versions of the **df** output.

```
suse$ df -h | perl -ane 'print join("\t", @F[0..4]), "\n"' > tmp1
suse$ df -i | perl -ane 'print join("\t", @F[0,1,4]), "\n"' > tmp2
suse$ join tmp1 tmp2
Filesystem   Size    Used    Avail   Use%  Inodes   IUse%
/dev/hda3    3.0G    1.9G    931M    68%   393216   27%
udev         126M    172K    126M    1%    32086    2%
/dev/hda1    92M     26M     61M     30%   24096    1%
/dev/hda6    479M    8.1M    446M    2%    126976   1%
…
```

A script version with no temporary files would look something like this:

```
#!/usr/bin/perl

for (split(/\n/, ‘df -h‘)) {
    @F = split;
    $h_part{$F[0]} = [ @F[0..4] ];
}

for (split(/\n/, ‘df -i‘) {
    @F = split;
    print join(“\t”, @{$h_part{$F[0]}}, $F[1], $F[4]), “\n”;
}
```

The truly intrepid can use **-i** in conjunction with **-pe** to edit files in place; Perl reads the files in, presents their lines for editing, and saves the results out to the original files. You can supply a pattern to **-i** that tells Perl how to back up the original version of each file. For example, **-i.bak** backs up **passwd** as **passwd.bak**. Beware—if you don't supply a backup pattern, you don't get backups at all. Note that there's no space between the **-i** and the suffix.

Add-on modules for Perl

CPAN, the Comprehensive Perl Archive Network at cpan.org, is the warehouse for user-contributed Perl libraries. Installation of new modules is greatly facilitated by the **cpan** command, which acts much like a **yum** or APT package manager dedicated to Perl modules. If you're on a Linux system, check to see if your distribution packages the module you're looking for as a standard feature—it's much easier to install the system-level package once and then let the system take care of updating itself over time.

On systems that don't have a **cpan** command, try running **perl -MCPAN -e shell** as an alternate route to the same feature:

```
$ sudo perl -MCPAN -e shell

cpan shell -- CPAN exploration and modules installation (v1.9205)
ReadLine support available (maybe install Bundle::CPAN or Bundle::CPANxxl?)

cpan[1]> install Class::Date
CPAN: Storable loaded ok (v2.18)
CPAN: LWP::UserAgent loaded ok (v5.819)
CPAN: Time::HiRes loaded ok (v1.9711)
… several more pages of status updates …
```

It's possible for users to install Perl modules in their home directories for personal use, but the process isn't necessarily straightforward. We recommend a liberal policy regarding system-wide installation of third-party modules from CPAN; the community provides a central point of distribution, the code is open to inspection, and module contributors are identified by name. Perl modules are no more dangerous than any other open source software.

Many Perl modules use components written in C for better performance. Installation involves compiling these segments, so you need a complete development environment including the C compiler and a full set of libraries.

As with most languages, the most common error found in Perl programs is the reimplementation of features that are already provided by community-written modules.[12] Get in the habit of visiting CPAN as the first step in tackling any Perl problem. It saves development and debugging time.

2.5 Python scripting

As projects become larger and more complex, the benefits of object-oriented design and implementation become clearer. Perl missed the OO boat by about five years, and although it paddled furiously to keep up, Perl's version of object-oriented programming still feels a bit hackish.

This section describes Python 2. Python 3 is in the works and is likely to be released during the lifetime of this book. But unlike Perl 6, it appears likely to be a relatively incremental update.

Engineers with a strong OO background usually like Python and Ruby, both scripting languages with a pronounced OO inflection. Python seems to be well onto the downhill side of the adoption curve at this point, so it's a relatively easy sale for management. Several operating systems, including OpenSolaris, are

12. Tom Christiansen commented, "That wouldn't be my own first choice, but it is a good one. My nominee for the most common error in programs is that they are usually never rewritten. When you take English composition, you are often asked to turn in an initial draft and then a final revision, separately. This process is just as important in programming. You've heard the adage 'Never ship the prototype.' Well, that's what's happening: people hack things out and never rewrite them for clarity and efficiency."

making major investments in Python scriptability. Ruby, by contrast, is still primarily associated with web development and is rarely used for general scripting.

Python was created by Guido van Rossum. It's easier to code and more readable than Perl. Python offers a simple-to-understand syntax that is easy to follow even if you didn't develop the code. If you're tired of remembering which comparison operators to use, you'll appreciate Python's unified approach. Python also offers additional data types that some system administrators find useful.

If Python is not already on your system, check your vendor's or distributor's list of available packages. It's an extremely common package and should be universally available. Failing that, you can get Python source code from python.org. That is also a central location for finding add-in modules developed by others.

For a more thorough introduction to Python than we can give here, Mark Pilgrim's *Dive Into Python* is a great place to start. It's available for reading or for download (without charge) at diveintopython.org, or as a printed book from Apress. A complete citation can be found on page 75.

Python quick start

As usual, we start with a quick "Hello, world!" script. As it happens, Python's "Hello, world!" is almost identical to Perl's.

```
#!/usr/bin/python
print "Hello, world!"
```

To get it running, set the execute bit or invoke the **python** interpreter directly:

```
$ chmod +x helloworld
$ ./helloworld
Hello, world!
```

This one-liner fails to illustrate Python's most scandalous break with tradition, namely, that indentation is logically significant. Python does not use braces, brackets, or `begin` and `end` to delineate blocks. Statements at the same level of indentation automatically form blocks. The exact indentation style (spaces or tabs, depth of indentation) does not matter. Python blocking is best shown by example, so here's an if-then-else statement:

```
#!/usr/bin/python

import sys

a = sys.argv[1]

if a == "1":
    print 'a is one'
    print 'This is still the then clause of the if statement.'
else:
    print 'a is', a
    print 'This is still the else clause of the if statement.'

print 'This is after the if statement.'
```

The third line imports the sys module, which contains the argv array. The then and else clauses both have two lines, each indented to the same level. The final print statement is outside the context of the if statement. As in Perl, Python's print statement accepts an arbitrary number of arguments. But unlike Perl, Python inserts a space between each pair of arguments and supplies a newline automatically. You can suppress the newline by including an extra comma at the end of the print line; the null argument tells print not to output the newline character.

Colons at the end of a line are normally a clue that the line introduces and is associated with an indented block that follows it.

```
$ python blockexample 1
a is one
This is still the then clause of the if statement.
This is after the if statement.

$ python blockexample 2
a is 2
This is still the else clause of the if statement.
This is after the if statement.
```

Python's indentation convention gives you less flexibility in the formatting of code, but it has the advantage of making code written by different people look the same, and it means that there is no need to sprinkle your code with pesky semicolons just to terminate statements.

Comments are introduced with a hash mark (#) and last until the end of the line, just as in **bash** and Perl.

You can split long lines by backslashing the end of line breaks. When you do this, only the indentation of the first line is significant. You can indent the continuation lines however you like. Lines with unbalanced parentheses, square brackets, or curly braces automatically signal continuation even in the absence of backslashes, but you can include the backslashes if doing so clarifies the structure of the code.

Some cut and paste operations convert tabs to spaces, and unless you know what you're looking for, this can drive you nuts. The golden rule is never to mix tabs and spaces; use one or the other for indentation. A lot of software makes the traditional assumption that tabs should fall at 8-space intervals, which is really too much indentation for readable code. Most in the Python community seem to prefer spaces and 4-character indentation.

However you decide to attack the indentation problem, most editors have options that can help save your sanity, either by outlawing tabs in favor of spaces or by displaying spaces and tabs differently. As a last resort, you can translate tabs to spaces with the **expand** command or use **perl -pe** to replace tabs with a more easily seen character string.

Objects, strings, numbers, lists, dictionaries, tuples, and files

All data types in Python are objects, and this gives them more power and flexibility than they have in Perl.

In Python, lists are enclosed in square brackets instead of parentheses. Arrays index from zero, which is one of the few concepts that doesn't change among the three scripting languages covered in this chapter.

New with Python are "tuples," which are essentially immutable lists. Tuples are faster than arrays and are helpful for representing data that should in fact be unmodifiable. The syntax for tuples is the same as for lists, except that the delimiters are parentheses instead of square brackets. Because (thing) looks like a simple algebraic expression, tuples that contain only a single element need an extra comma to disambiguate them: (thing,).

Here's some basic variable and data type wrangling in Python:

```
#!/usr/bin/python

name = 'Gwen'
rating = 10
characters = [ 'SpongeBob', 'Patrick', 'Squidward' ]
elements = ( 'lithium', 'carbon', 'boron' )

print "name:\t%s\nrating:\t%d" % (name, rating)
print "characters:\t%s" % characters
print "elements:\t%s" % (elements, )
```

This example produces the following output:

```
$ python objects
name:         Gwen
rating:       10
characters:   ['SpongeBob', 'Patrick', 'Squidward']
elements:     ('lithium', 'carbon', 'boron')
```

Variables in Python are not syntactically marked or declared by type, but the objects to which they refer do have an underlying type. In most cases, Python does not automatically convert types for you, but individual functions or operators may do so. For example, you cannot concatenate a string and a number (with the + operator) without explicitly converting the number to its string representation. However, formatting operators and statements do coerce everything to string form. Every object has a string representation.

The string formatting operator % is a lot like the sprintf function from C or Perl, but it can be used anywhere a string can appear. It's a binary operator that takes the string on its left and the values to be inserted on its right. If there is more than one value to insert, the values must be presented as a tuple.

A Python dictionary is the same thing as a Perl hash; that is, a list of key/value pairs. Dictionary literals are enclosed in curly braces, with each key/value pair being separated by a colon.

```
#!/usr/bin/python

ordinal = { 1 : 'first', 2 : 'second', 3 : 'third' }
print "The ordinal array contains", ordinal
print "The ordinal of 1 is", ordinal[1]
```

In use, Python dictionaries are a lot like arrays, except that the subscripts (keys) can be objects other than integers.

```
$ python dictionary
The ordinal array contains {1: 'first', 2: 'second', 3: 'third'}
The ordinal of 1 is first
```

Python handles open files as objects with associated methods. True to its name, the readline method reads a single line, so the example below reads and prints two lines from the **/etc/passwd** file.

```
#!/usr/bin/python

f = open('/etc/passwd', 'r')
print f.readline(),
print f.readline(),
f.close()
```

```
$ python fileio
at:x:25:25:Batch jobs daemon:/var/spool/atjobs:/bin/true
bin:x:1:1:bin:/bin:/bin/true
```

The trailing commas are in the print statements to suppress newlines because each line already includes a newline character as it is read from the original file.

Input validation example

The scriptlet below is the Python version of our by-now-familiar input validator. It demonstrates the use of subroutines and command-line arguments along with a couple of other Pythonisms.

```
#!/usr/bin/python

import sys
import os

def show_usage(message, code = 1):
    print message
    print "%s: source_dir dest_dir" % sys.argv[0]
    sys.exit(code)
```

```
if len(sys.argv) != 3:
    show_usage("2 arguments required; you supplied %d" % (len(sys.argv) - 1))
elif not os.path.isdir(sys.argv[1]):
    show_usage("Invalid source directory")
elif not os.path.isdir(sys.argv[2]):
    show_usage("Invalid destination directory")

source, dest = sys.argv[1:3]

print "Source Directory is", source
print "Destination Directory is", dest
```

In addition to importing the sys module, we also import the os module to gain access to the os.path.isdir routine. Note that import doesn't shortcut your access to any symbols defined by modules; you must use fully qualified names that start with the module name.

The definition of the show_usage routine supplies a default value for the exit code in case the caller does not specify this argument explicitly. Since all data types are objects, function arguments are passed by reference.

The sys.argv array contains the script name in the 0 position, so its length is 1 greater than the number of command-line arguments that were actually supplied. The form sys.argv[1:3] is an array slice. Curiously, slices do not include the element at the far end of the specified range, so this slice includes only sys.argv[1] and sys.argv[2]. You could simply say sys.argv[1:] to include the second and subsequent arguments.

Like both **bash** and Perl, Python has a dedicated "else if" condition; the keyword is elif. There is no explicit case or switch statement.

The parallel assignment of the source and dest variables is a bit different from the Perl version in that the variables themselves are not in a list. Python allows parallel assignments in either form.

Python uses the same comparison operators for numeric and string values. The "not equal" comparison operator is !=, but there is no unary ! operator; use not for this. The Boolean operators and and or are also spelled out.

Loops

The fragment below uses a for…in construct to iterate through the range 1 to 10.

```
for counter in range(1, 10):
    print counter,
```

As with the array slice in the previous example, the right endpoint of the range is not actually included. The output includes only the numbers 1 through 9:

```
1 2 3 4 5 6 7 8 9
```

This is Python's only type of for loop, but it's a powerhouse. Python's for has several features that distinguish it from for in other languages:

- There is nothing special about numeric ranges. Any object can support Python's iteration model, and most common objects do. You can iterate through a string (by character), a list, a file (by character, line, or block), an array slice, etc.
- Iterators can yield multiple values, and you can have multiple loop variables. The assignment at the top of each iteration acts just like Python's regular multiple assignments.
- Both for and while loops can have else clauses at the end. The else clause is executed only if the loop terminates normally, as opposed to exiting through a break statement. This feature may initially seem counterintuitive, but it handles certain use cases quite elegantly.

The example script below accepts a regular expression on the command line and matches it against a list of Snow White's dwarves and the colors of their dwarf suits. The first match is printed with the portions that match the regex surrounded by underscores.

```
#!/usr/bin/python

import sys
import re

suits = { 'Bashful':'red', 'Sneezy':'green', 'Doc':'blue', 'Dopey':'orange',
     'Grumpy':'yellow', 'Happy':'taupe', 'Sleepy':'puce' }
pattern = re.compile("(%s)" % sys.argv[1])

for dwarf, color in suits.items():
     if pattern.search(dwarf) or pattern.search(color):
          print "%s's dwarf suit is %s." % \
               (pattern.sub(r"_\1_", dwarf), pattern.sub(r"_\1_", color))
          break
else:
     print "No dwarves or dwarf suits matched the pattern."
```

Here's some sample output:

```
$ python dwarfsearch '[aeiou]{2}'
Sn_ee_zy's dwarf suit is gr_ee_n.

$ python dwarfsearch go
No dwarves or dwarf suits matched the pattern.
```

The assignment to suits demonstrates Python's syntax for encoding literal dictionaries. The suits.items() method is an iterator for key/value pairs—note that we're extracting both a dwarf and a suit color on each iteration. If you only wanted to iterate through the keys, you could just say for dwarf in suits.

Python implements regular expression handling through its re module. No regex features are built into the language itself, so regex-wrangling with Python is a bit clunkier than with Perl. Here, the regex pattern is initially compiled from the first command-line argument surrounded by parentheses to form a capture group. Strings are then tested and modified with the search and sub methods of the regex object. You can also call re.search et al. directly as functions, supplying the regex to use as the first argument. The \1 in the substitution string is a back-reference to the contents of the first capture group.

2.6 Scripting best practices

Although the code fragments in this chapter contain few comments and seldom print usage messages, that's only because we've skeletonized each example to make specific points. Real scripts should behave better. There are whole books on best practices for coding, but here are a few basic guidelines:

- When run with inappropriate arguments, scripts should print a usage message and exit. For extra credit, implement --**help** this way, too.
- Validate inputs and sanity-check derived values. Before doing an **rm -rf** on a calculated path, for example, you might have the script double-check that the path conforms to the pattern you expect. You may find your scripting language's "taint" feature helpful.
- Return an appropriate exit code: zero for success and nonzero for failure. Don't feel compelled to give every failure mode a unique exit code, however; consider what callers will actually want to know.
- Use appropriate naming conventions for variables, scripts, and routines. Conform to the conventions of the language, the rest of your site's code base, and most importantly, the other variables and functions defined in the current project. Use mixed case or underscores to make long names readable.[13]
- Use variable names that reflect the values they store, but keep them short. number_of_lines_of_input is way too long; try n_lines.
- Consider developing a style guide so that you and your colleagues can write code according to the same conventions. A guide makes it easier for you to read other people's code and for them to read yours.
- Start every script with a comment block that tells what the script does and what parameters it takes. Include your name and the date. If the script requires nonstandard tools, libraries, or modules to be installed on the system, list those as well.

13. The naming of the scripts themselves is important, too. In this context, dashes are more common than underscores for simulating spaces, as in **system-config-printer**.

- Comment at the level you yourself will find helpful when you return to the script after a month or two. Some useful points to comment on are the following: choices of algorithm, reasons for not doing things in a more obvious way, unusual paths through the code, anything that was a stumbling block during development. Don't clutter code with useless comments; assume intelligence and language proficiency on the part of the reader.
- Code comments work best at the granularity of blocks or functions. Comments that describe the function of a variable should appear with the variable's declaration or first use.
- It's OK to run scripts as root, but avoid making them setuid; it's tricky to make setuid scripts completely secure. Use **sudo** to implement appropriate access control policies instead.
- With **bash**, use **-x** to echo commands before they are executed and **-n** to check commands for syntax without executing them.
- Perl's **-w** option warns you about suspicious behaviors such as variables used before their values are set. You can include this option on a script's shebang line or turn it on in the program's text with `use warnings`.
- In Python, you are in debug mode unless you explicitly turn it off with a **-0** argument on the command line. That means you can test the special `__debug__` variable before printing diagnostic output.

Tom Christiansen suggests the following five Golden Rules for producing useful error messages:

- Error messages should go to STDERR, not STDOUT.
- Include the name of the program that's issuing the error.
- State what function or operation failed.
- If a system call fails, include the **perror** string ($! in Perl).
- Exit with some code other than 0.

Perl makes it easy to follow all five rules:

```
die "can't open $filename: $!";
```

2.7 Recommended reading

BROOKS, FREDERICK P., JR. *The Mythical Man-Month: Essays on Software Engineering.* Reading, MA: Addison-Wesley, 1995.

Shell basics and bash scripting

ALBING, CARL, JP VOSSEN, AND CAMERON NEWHAM. *Bash Cookbook*. Sebastopol, CA: O'Reilly Media, 2007.

KERNIGHAN, BRIAN W., AND ROB PIKE. *The UNIX Programming Environment*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

NEWHAM, CAMERON, AND BILL ROSENBLATT. *Learning the bash Shell (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2005.

POWERS, SHELLEY, JERRY PEEK, TIM O'REILLY, AND MIKE LOUKIDES. *Unix Power Tools, (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2002.

Regular expressions

FRIEDL, JEFFREY. *Mastering Regular Expressions (3rd Edition)*, Sebastopol, CA: O'Reilly Media, 2006.

GOYVAERTS, JAN, AND STEVEN LEVITHAN. *Regular Expressions Cookbook*. Sebastopol, CA: O'Reilly Media, 2009.

Perl scripting

WALL, LARRY, TOM CHRISTIANSEN, AND JON ORWANT. *Programming Perl (3rd Edition),* Sebastopol, CA: O'Reilly Media, 2000.

SCHWARTZ, RANDAL L., TOM PHOENIX, AND BRIAN D FOY. *Learning Perl (5th Edition)*, Sebastopol, CA: O'Reilly Media, 2008.

BLANK-EDELMAN, DAVID. *Automating System Administration with Perl*, Sebastopol, CA: O'Reilly Media, 2009.

CHRISTIANSEN, TOM, AND NATHAN TORKINGTON. *Perl Cookbook (2nd Edition).* Sebastopol, CA: O'Reilly Media, 2003.

Python scripting

BEAZLEY, DAVID M. *Python Essential Reference (4th Edition)*, Reading, MA: Addison-Wesley, 2009.

GIFT, NOAH, AND JEREMY M. JONES. *Python for Unix and Linux System Administrators*, Sebastopol, CA: O'Reilly Media, 2008.

MARTELLI, ALEX, ANNA MARTELLI RAVENSCROFT, AND DAVID ASCHER. *Python Cookbook (2nd Edition)*, Sebastopol, CA: O'Reilly Media, 2005.

PILGRIM, MARK. *Dive Into Python.* Berkeley, CA: Apress, 2004. This book is also available for free on the web at diveintopython.org.

2.8 Exercises

E2.1 UNIX allows spaces in filenames. How do you find files whose names contain embedded spaces? How do you delete them? Do **bash**, Perl, and Python handle spaces in filenames gracefully, or do you need to take special precautions? Outline appropriate rules of thumb for scripting.

E2.2 Write a simple **bash** script (or pair of scripts) to back up and restore your system.

E2.3 Using regular expressions, write a Perl or Python script to parse a date in the form produced by the **date** command (e.g., Tue Oct 20 18:09:33 PDT 2009) and determine whether it is valid (e.g., no February 30th, valid time zone, etc.). Is there an off-the-shelf library or module that lets you do this in one line? If so, explain how to install it and recode your script to use it.

E2.4 Write a script that enumerates the system's users and groups from **/etc/passwd** and **/etc/group** (or their network database equivalents). For each user, print the user's UID and the groups of which the user is a member.

E2.5 Refine the get_string example on page 63 to accept only integers. It should accept three parameters: the prompt string, a lower limit on the acceptable integers, and an upper limit on the acceptable integers.

E2.6 Find an undocumented script that's used in your environment. Read it and make sure you understand its function. Add comments and write a man page for the script.

☆ E2.7 Write a script that displays a one-screen summary of status data related to one of the following categories: CPU, memory, disk, or network. The script should leverage OS commands and files to build an easy-to-understand dashboard that includes as much information as possible.

☆ E2.8 Build a menu-driven interface that makes it easy to select command-line options for **top**, **sar**, or the performance analysis tool of your choice.

☆ E2.9 Write a script to test a server's network connectivity and the upstream services on which it depends (e.g., DNS, file service, LDAP or other directory service). Have it send you email or a text message if problems are discovered.