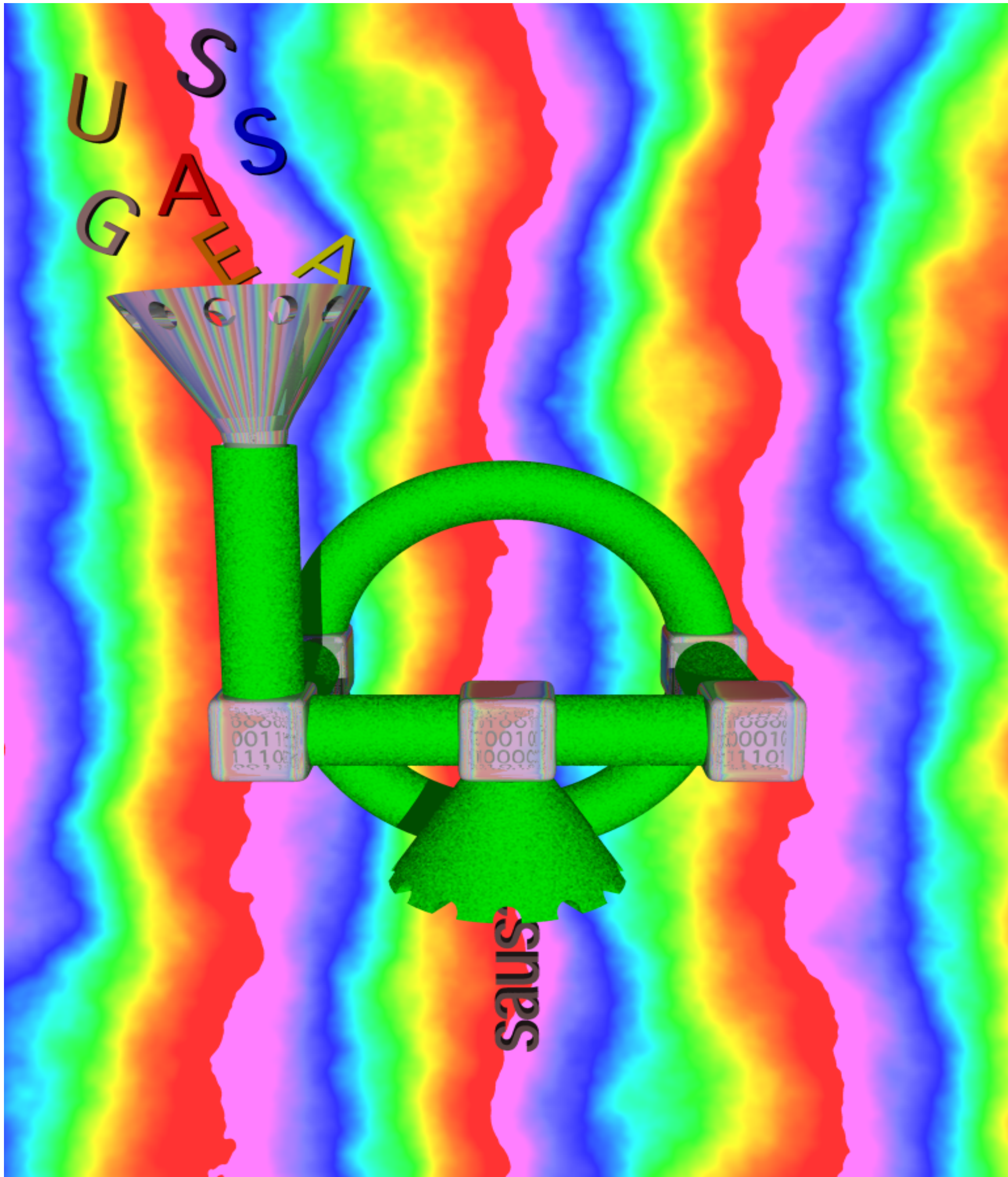
Perls Before Swine



Copyright © 2012 Jerry Stratton

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1. A copy of the license is included in the section entitled "GNU Free Documentation License"

January 11, 2012

Introduction 1
What is HTML? 1
The web site 1
What do you need? 2
Sample Data 2
Text Editor 2
Terminal or Shell 2
The basic Perl filter 3
What is it doing? 3
Indentation 4
Basic regular expressions 5
Splitting and printing 6
Comments 7
Smarter scripts 9
Capturing errors 9
Help! 10
Command-line switches 11
Case sensitivity 13
Boolean logic 14
Exiting loops ahead of time 14
Multiple options 16
Script confusion 17
The current script 18
Arrays and functions 21
Sort numerically 23
A smarter join 24
Format conversions 26
The current script 26
Custom search 29
New fields 31
Custom sort 31
Backquoting special characters 35
The current script 36
Creating files 39
Creating folders 42
Replacing text 43
Try to break it 45
Timestamps 46
The current script 48
SQL database 53
Installing from CPAN 53
Using SQLite 54
Creating tables 55
Inserting data 56
The final script 58
Web CGIs 65
Reference 69
Boolean logic 69
Comparison operators 69
File tests 70
Regular expressions 70
SilverService 71
More Information 73
GNU Free Documentation License 73

Introduction

What is Perl?

Perl is a twenty-year-old scripting language designed for managing text. It is cross-platform, running on Linux, Unix, Mac OS X, Windows, and probably many more operating systems. It comes pre-installed on most operating systems today, and is used for managing server tasks, formatting documents, and filtering data. It may be the most-used programming language on the web, and has sometimes been called the duct-tape of the Internet.

If you're familiar with the use of duct tape, you'll have an idea of what Perl is used for. Perl is not the prettiest of solutions. But it works. It holds together things that would otherwise never hold together, and is a useful tool for creating quick solutions to thorny problems. There is an elegance in duct tape, an elegance in the solutions of the trenches. When something is broken it needs to be fixed.

The web site

You can find the latest version of this tutorial, as well as the resources archive, at http://www.hoboes.com/NetLife/Swine/.

Why use Perl?

If you're a web page designer and you're interested in programming, for example, you're probably already using PHP. What makes Perl useful instead of PHP? The answer is the command line. Perl excels as filter and as glue. It is great at taking some input—usually text—and modifying it. It acts as a great text sausage machine, grinding up text and spitting it out.

Perl also makes for a great glue tying together the various command line programs you use and automating your use of them. Perl scripts are often used as cron jobs, running automatically at specified times. Perl is a great way of taking what you want to give your command line program and converting it into what the command line program expects. It is great at mediating between two or more data sources.

If you manage a web site or a MySQL database and need regular backups and monitoring, or if you need to regularly collect and collate data from a set of files, Perl is a great tool to know. If your task is a series of changes—if you can think of it as a series of sieves or as an assembly line of tasks—Perl can provide rapid automation for that task.

What do you need?

Sample Data

Go to http://www.hoboes.com/NetLife/Swine/ to download the resources archive. Inside, you'll find a text file called "songs.txt". We'll be using that in this tutorial.

Text Editor

You will need a text editor, such as Smultron on the Macintosh or NoteTabPro on Windows. If you intend to edit Perl scripts directly on a remote server, you will need familiarity with a Unix text editor such as vi or pico.

If you're using a GUI text editor, you'll want to make sure that it saves your files with Unix line endings. It usually won't matter, but it can sometimes help track down errors.

Terminal or Shell

You will need to be able to execute your scripts. Normally you will do this from some sort of terminal or shell application. If you are running these scripts on a remote server, you will probably use ssh, or secure shell to get to that server. If you are running them on your local Mac OS X workstation, you'll use the Terminal application in your Utilities folder.

The basic Perl filter

Make sure you've downloaded the sample data, and then create the following text file:

```
#!/usr/bin/perl
while (<>) {
	print;
}
```

Save this file as the filename *show*. Once you've saved it, make sure that it is *executable* by you. Go to your command line, make sure that your are in the correct folder, and type:

```
chmod u+x show
```

On Mac OS X, you can ensure that you are in the correct folder by typing "cd" in the terminal, a space, and then dragging the folder onto your terminal window. Press return, and you will *c*hange *d*irectory into that folder.

Now, type:

```
./show show
```

The script should show you itself. Make sure that songs.txt is in the same directory as your script, and type:

```
./show songs.txt
```

The show script should show you all 7,006 lines in the songs.txt file.

What is it doing?

This is about as simple of a Perl script as you can get. While it doesn't do much yet, this is a shell around which you can build quite a few useful scripts.

```
#!/usr/bin/perl
```

The first line is not Perl. The first line tells the operating system what language this script is written in. More specifically, it tells the operating system which program can *interpret* this script.

Most shell scripts use the pound character ("#") for *comments*. What it really means is that every line that begins with a pound character is ignored by the scripting language. So Perl ignores this line because it begins with a pound character, but the operating system or shell that you're using knows to send this script off to the program called /usr/bin/perl.

If your computer didn't come with Perl pre-installed, you may have it installed in /usr/local/bin/perl instead. Nowadays, however, most operating systems come with Perl pre-installed.

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

This is a *while block*. The part between the parentheses is an *expression* and the part between the two curly brackets gets acted on for as long as that expression does not return false, empty, zero, or, basically, as long as it returns *something*. We can put as much stuff as we want between those curly brackets. Perl will repeat them, or *loop* through them, for as long as that expression gives it something to work on.

The expression we have here is “<>”. This tells Perl that we want, line by line, everything in the files that we mentioned on the command line. If we don’t mention any files on the command line, Perl takes the *standard input* and gives that to us line by line instead.

For example, type:

```
./show
```

When you press return, you won’t get the command line back. Because we didn’t specify any filenames on the command line, Perl is waiting for the standard input. Because we didn’t give it any, it is waiting for us to type it. Type a few lines, pressing return after each line, and you’ll see the script echo whatever you type back.

Type Control-D to exit. Then type:

```
echo "Now is the time for all good muskrats to come to the aid of their country" | ./show
```

In Unix, the vertical bar is the *pipe* character. Whatever is on the left gets *piped* through to whatever is on the right. Echo echoes text to the screen normally, but in the above command the output from echo gets piped through to the show script.

Finally:

```
print;
```

This command is perhaps the most common one in Perl. You use it to *output* something, either to the screen or to a file.

What is it printing? Perl often makes assumptions about what you want. When we don’t give print anything to print, Perl assumes we want to print the current line from the *while* loop.

So what this script does is go through every line it gets and prints it out to the screen. If you’re familiar with Unix, we’ve just reinvented the cat command.

Indentation

In the above script, the *print* command is indented by one tab. Indentation makes it much easier to read your scripts. It is much easier to see where *where* blocks and other blocks begin and end if those blocks are indented. While Perl does not care about indentation, it is for all practical purposes required that you indent; if you have blocks inside of blocks, those will be indented further.

Basic regular expressions

Our current script is a *filter*. It takes some raw data on one end, filters it, and produces modified data on the other end. You can also think of a filter as a sausage grinder. Our filter, however, is a very leaky sieve: it currently lets everything through.

One of the things that Perl does very well is to filter what it gets according to a *regular expression*. A regular expression is a very versatile form of searching. Change the print line to:

```
print if /Mellow/;
```

And then type:

```
./show songs.txt
```

You should see all of the songs in Donovan's *Mellow Yellow* album as well as a few songs by the *Mellow Men*.

In Perl, everything between two slashes, like that, is a regular expression. Since we didn't tell Perl what text we want the regular expression to apply to, it assumes we want to apply it to the current line.

Any command or function followed by "if *expression*" will be performed only if that expression returns something. If it returns nothing (as it will in this case when the line does not contain Mellow), that line does not get performed. In this case, that line does not get printed.

It would be annoying to have to edit our script every time we wanted to look for some text in our file. So we can modify the script to look on the command line for the text we want. Add one line to the script, so that it now reads:

```
#!/usr/bin/perl
$searchFor = shift;
while (<>) {
	print if /$searchFor/;
}
```

Let's look for all mentions of *Yellow* in the song listing:

```
./show Yellow songs.txt
```

You should see a bunch of songs from Elton John's Goodbye *Yellow Brick Road* and Donovan's *Mellow Yellow*, and a few versions of Joni Mitchell's *Big Yellow Taxi*.

We'll talk more specifically about grabbing stuff from the command line in a bit, but the *shift* command grabs the first item off of a list of items. If you don't specify a list of items, it assumes you want the list of items that were on the command line.

On the left of *shift*, we have "$searchFor =". In Perl, as in many scripting languages, the "=" is used to assign values to variables. Variables that can contain individual values (such as the word "Yellow" or the phrase "Voices in my head") are called *scalars* in Perl, and they always begin with a dollar sign. So this line takes the first item on the command line, *shifts* it out of the list of items on the command line (so that it is no longer in that list) and assigns it to the variable called *$searchFor*.

Let's talk a little more about regular expressions first, though, because they are so useful. Do a search for "Voices in my head":

```
./show "Voices in my head" songs.txt
```

On the command line, "arguments"—the things you pass to your scripts—are separated by spaces. If you want those spaces to be part of your argument, you need to surround the argument with quotes. If we didn't have the quotes around "Voices in my head", Perl would think we wanted to search for Voices in four files: "in", "my", "head", and "songs.txt".

However, the show script still doesn't show anything. Try:

```
./show Voices songs.txt
```

There are several songs, and one of them is the one we're looking for. However, it is slightly different from what we typed: it has upper case letters where we typed lower case letters.

Rather than have to remember the exact case for every song title, we can tell Perl to ignore the case of what we're looking for. We can tell it to be *case insensitive*. It's very simple: just add an "i" after the final slash on the print line:

```
print if /$searchFor/i;
```

Once you've done that, repeat the "Voices in my head" search, and you will see the one song that has that phrase in its title.

That's the form for regular expressions: a slash, something to search for, another slash, and single-letter options to modify the search. Regular expressions will get a lot more complicated than that as you learn more Perl, but that's the basic form that they will take.

Splitting and printing

Our search is working nicely, but it is returning a lot more information than we really need. The data file contains quite a bit of information about our songs. It'd be nice to display it in a more useful form. To do this, we first need to break it up into its pieces.

Looking at the data, it looks like each line consists of a song title, a duration, an artist name, an album name, a year, some number from 0 to 100, a timestamp, its position in the tracks on the album, and a genre. Each of these items is separated by a tab character. Replace the *print* line with the following two lines:

```
($song, $duration, $artist, $album) = split("\t");
print "$song ($album, by $artist)\n" if /$searchFor/i;
```

Both of these new lines illustrate some important features of Perl, so let's take them piece by piece:

```
($song, $duration, $artist, $album) = split("\t");
```

The "=" character means that we're doing an assignment here. Whatever happens on the right is going to get assigned to the stuff we have on the left. The right is the *split* function. This function splits a piece of text into pieces, based on another piece of text. The text we're splitting on is the tab character. In Unix, the tab is often specified using backslash-t, or "\t". The text we're splitting is the

current line, because we didn't tell Perl what text to split. So, if the current line has eight tabs, this is going to split it into nine pieces.

On the left, we have a list of variables. They all begin with dollar signs, so we know that they are all expecting a single piece. Perl is going to take those pieces we generated on the right and assign them, in order to the variables we've specified on the left. If there are more pieces on the right than on the left, the extra pieces will be ignored.

So when this line is done, we should have a variable containing the song title, duration of the song, the artist's name, and the album's name. We'll ignore everything else for now.

```
print "$song ($album, by $artist)\n" if /$searchFor/i;
```

Up until now, we've been letting Perl print out the current line as soon as it comes through. Our new print command actually has something to print. We're telling print to print a piece of text consisting of the variable $song, a space, a parentheses, the variable $album, a comma, a space, the word "by", another space, the variable $artist, a closing parentheses, and then a "new line" character. Just as we saw for tabs, the new-line character has a special code beginning with a backslash: "\n".

In Perl, most text, whether your are printing it or assigning to a variable, will be surrounded by quotes. If you surround text with double-quotes, any variables inside that text will be "interpreted" and replaced with their values. If you use single-quotes, Perl will not search the text for variables.

So, go ahead and try the new version of *show*:

```
./show Voices songs.txt
```

You should see, in a much more readable form, a skit by George Carlin, songs from Nanci Griffith's *Other Voices, Other Rooms*, and several songs that contain the word "voices".

Comments

I mentioned earlier that the pound sign at the beginning of a line causes Perl to completely ignore that line. This makes the pound sign a useful way of adding *comments* to your scripts. Comments are very important: they help you remember what you meant by this snippet of script several months or even years later when you look at the script again.

This script, for example, might be commented as follows:

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre

#the first item on the command line is what we're searching for
$searchFor = shift;
while (<>) {
	#split out the song, duration, artist, and album from the current line
	($song, $duration, $artist, $album) = split("\t");

	#print song information if this line contains our search text
```

```
		print "$song ($album, by $artist)\n" if /$searchFor/i;
	}
```

You don’t need to comment every line, but it is a good idea to comment every section. You’ll usually want to put a comment in front of any *while* block, or other large block of Perl lines.

Smarter scripts

Capturing errors

It isn't that difficult to trip up the script we've got so far. If you just type ./show and press return, not only does it wait on the command line for us to type something, it doesn't even know that we didn't tell it to search for anything.

Often, when you can identify command line arguments that you know are wrong, you will want to check for those arguments, and print instruction text when someone types something unexpected.

In this case, if there is nothing to search for, the person using the script probably doesn't know how to use the script. We can tell them how to use it. Change the script by adding an "if" line above the "while", indenting everything, and then adding several lines at the bottom:

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre

#the first item on the command line is what we're searching for
if ($searchFor = shift) {
	while (<>) {
		#split out the song, duration, artist, and album
		($song, $duration, $artist, $album) = split("\t");

		#print the information if this line contains our search text
		print "$song ($album, by $artist)\n" if /$searchFor/i;
	}
} else {
	help();
}

#describe how this script is used
sub help {
	print "Syntax: show <search text> [song files]\n";
	print "\tSearch for <search text> in the song file. If no song file is specified\n";
	print "\t'show' will expect it on standard input.\n";
	print "\tA song file is a tab-delimited file with:\n";
	print "\ttitle, duration, artist, album, year, rating, rip date, track position, genre\n";
}
```

The word "if" starts a block very much like the word "while" does. Unlike *while*, however, an *if* block is only performed once. Otherwise it is very similar. If the expression inside the parentheses of the *if* line returns something, the *if* block is performed. Otherwise, it isn't. Some *if* blocks have a corresponding *else* block. If so, the *else* block is only performed if the *if* block is *not* performed.

Notice how the *while* block is indented further beyond the indentation of the *if* block. I indented it further once I placed the *if* block around it. You should do so also. As your scripts become more and more complex, failure to indent will make it practically impossible to fix errors.

The word “sub” also starts a block. Unlike *while* and *if*, however, a *sub* block is never performed unless asked. The word that follows *sub* is the *name* of this subroutine. It is how we ask Perl to perform this block. Anywhere where we have that name followed by two parentheses, Perl will perform the *sub* block corresponding to that name.

In our case, if the *shift* does *not* assign something into $searchFor, we call the *help* subroutine. The term *subroutine* is somewhat archaic. We almost never use the term *routine* anymore, and even *subroutine* is fading from use. But that’s the origin of the word *sub* to mark these blocks of Perl lines.

Help!

One of the advantages of subroutines is that partitioning off some Perl lines allows us to call those lines from multiple places without having to retype the lines. This improves the readability of our script and also the reliability. If we make a mistake in the subroutine, we can fix it in the subroutine.

We’ve got this subroutine called *help* but currently the only way to see it is to do something wrong. It might be nice to ask for the help without having to do something wrong.

When we want to alter the way a program works from the command line, we usually use *switches*. In Unix, switches usually begin with a single dash if they are a single character, or double dashes if they are a word. We’ll use words here just to make them easier to read. For example, to display the “help” message, we might use “./show –help”.

Add the following five lines above the “first item on the command line” comment:

```
#if they ask for help, do it and exit
if ($ARGV[0] eq "--help") {
	help();
	exit;
}
```

Now, type:

```
./show --help
```

And you should see the help message displayed. It doesn’t matter what else you type on the command line, as long as the first argument is “--help” you’ll get the help message and that’s it.

The important new section is the one that checks $ARGV[0]. @ARGV is the list of all command-line arguments. In Perl, lists—often called arrays, or simple arrays—begin with the @ character. If, however, you want an item *in* the array, you preface it with the dollar sign.

$ARGV[0] is the first item in the list called ARGV. Perl, like many programming languages, starts counting from zero rather than from one. The first item in a list is item 0, not item 1. The second item is $ARGV[1] (if there is one), and so on.

What we're checking is whether or not the first argument is *equal to* "--help". If it is, we call the *help* subroutine and then *exit*. In Perl, *exit* will end the script completely. It doesn't matter what else comes after the exit line, Perl ends the script and returns you to the command line.

Finally, don't forget to add a line to the help text describing how to get help:

```
print "\t--help: print this help text\n"
```

You'll always want to update your help subroutine whenever you add new features to your script, or modify existing features.

Command-line switches

So now we have a script with one command-line switch, but switches are like potato chips: once you start, you can't have just one. Here's an example: we've currently made our script case-insensitive, so that we don't have to worry about remembering the exact case of the text we're looking for. But what if we *want* it to be case-sensitive? Let's add a *case* switch.

To do this, though, we're going to need to "generalize" our search for command-line switches. If we just have a series of *if*s, that will mean that either we can't have more than one command-line switch, or we have to put them in an exact order. That will always be too difficult to remember, especially when we have eight or more switch possibilities.

One way of doing this is to loop through the beginning arguments as long as the argument is a switch. Stop looping when it is no longer a switch. We can use a *while* block for this. Replace our *help* switch's five lines with:

```
#strip off the command-line switches and act on or remember them
while ($ARGV[0] =~ /^--(.+)/) {
	$switch = $1;

	#pull this switch off of the front of the list
	shift;

	#if they ask for help, do it and exit
	if ($switch eq "help") {
		help();
		exit;
	}
}
```

This snippet does the same thing as the previous "help-only" snippet, but it will allow us to add any switches we want.

```
while ($ARGV[0] =~ /^--(.+)/) {
```

The "=~" is new. It is used to match a scalar variable against a regular expression. The variable goes on the left, and the regular expression goes on the right.

And what a regular expression! Let's take it piece by piece.

It begins with a caret, or “hat” character. The caret marks the beginning of a piece of text. Whatever comes next in the regular expression will only match if it comes at the beginning of the text. So, since the next two characters are two dashes, two dashes will only match if the two dashes are at the beginning. This differs from our previous regular expressions, where the text we specified could occur anywhere on the line.

After the two dashes, we have “(.+)”. The parentheses are easy: they tell Perl to remember that part of the match, whatever it is. We’ll see what that means in a moment.

The period, or “dot”, matches a single character. It can be *any* character.

The plus sign matches one or more of the previous piece of the regular expression. The previous piece is the dot, so the dot and the plus means one or more of any character.

Taken as a whole, this regular expression will match --help, --switch, --q, --rain, or even --planet-99x. It will *not* match help--, switch--station, or anything else that does not begin with two dashes.

```
$switch = $1;
```

The next line assigns the value of the variable $1 to the variable $switch. After a regular expression, Perl remembers any items in parentheses, and it remembers them by putting them into $1, $2, $3, $4, etc., on up to however many sets of parentheses were in the regular expression. We only have one set of parentheses, so we only get $1.

Because our parentheses were after the two dashes, $switch will now contain the part of that switch not including the initial two dashes.

```
#pull this switch off of the front of the list
shift;
```

We’ve already used *shift*. It shifts an item off of the front of a list. By default, it shifts it off of the front of the list of command-line arguments. Since @ARGV is the list of arguments, *shift* shifts the first argument off of @ARGV. Once shifted off, that first argument is gone. What used to be the second argument is now the first argument. This gets us ready for the next turn through the loop. $ARGV[0] is now the next argument.

```
#if they ask for help, do it and exit
if ($switch eq "help") {
	help();
	exit;
}
```

This section looks very familiar. The only part that’s changed is the *if* line. Instead of checking to see if $ARGV[0] is equal to “--help”, we’re checking to see if $switch is equal to “help”. If it is, we call the *help* subroutine and exit the script.

Case sensitivity

So after all that, we still haven't added case sensitivity to the script. But now, we can add pretty much anything we want. Our switch is going to be called *case*. What we'll do is set $sensitive if we want the search to be case sensitive.

Replace the closing curly bracket where we're checking for the *help* option with:

```
} elsif ($switch eq "case") {
        $sensitive = 1;
}
```

The whole switch section should look like:

```
if ($switch eq "help") {
        help();
        exit;
} elsif ($switch eq "case") {
        $sensitive = 1;
}
```

This is pretty simple, so far. If the command-line switch is "--case" then assign the number 1 to the variable $sensitive.

Replace the line that prints out the song information with:

```
if ($sensitive) {
        $matched = /$searchFor/;
} else {
        $matched = /$searchFor/i;
}
#print the information if this line contains our search text
print "$song ($album, by $artist)\n" if $matched;
```

If $sensitive has something in it, we match without the case-insensitivity modifier. Otherwise, we match with the case-insensitivity modifier. We assign the result of that match to $matched, and then if $matched has something in it we print the song information. (You might ask why we don't just create a variable that either has "i" in it or not. The answer is that we can't do that. A variable won't work in that part of a regular expression.)

Now, you can search for:

```
./show --case Yellow songs.txt
./show --case yellow songs.txt
```

And get different results. Add the help line to the help subroutine and we're done with this option:

```
print "\t--case: be sensitive to upper and lower case\n";
```

Boolean logic

So far our search has been for things that match. But sometimes filters are useful to filter *out* rather than filter *in*. We can add a switch that will cause the script to print out only those songs that *don't* match the search. To do this, we're going to need to understand Boolean logic. In case you've forgotten your Boolean logic from high school, it is basically about *true* and *false*. Perl treats items that contain something other than zero as *true*. It treats items that contain zero or nothing as *false*.

Code in your *if* or *while* parentheses are treated by Perl as Boolean expressions, as true or false.

You can reverse something from false to true or true to false with NOT, which in Perl is the exclamation point.

First, add the switch to our list of switches:

```
} elsif ($switch eq "reverse") {
	$reverse = 1;
}
```

Then, add some new code to the while loop, after we assign a match or lack thereof to $match, but before we print the song information:

```
#reverse the match if we want non-matching lines
if ($reverse) {
	$matched = !$matched;
}
```

You might also want to change the comment in front of the *print* line:

```
#print the information if this line is one we want
```

And, of course, add a line to the help subroutine:

```
print "\t--reverse: filter out songs that contain the search text\n";
```

So, now, if you want to see every song that does not mention *best* anywhere, use:

```
./show --reverse best songs.txt
```

Well. That showed a lot. Let's see if we can do something about that.

Exiting loops ahead of time

Often when we're testing we don't really want to see *everything*, we just want to see that it worked. The first several results will let us know that. Let's put in a switch to limit the number of results to a specified maximum. We'll call this switch *limit* and it will be followed by a number.

```
} elsif ($switch eq "limit") {
	$limit = shift;
	if ($limit !~ /^[1-9][0-9]*$/) {
		print "\nYou must limit to a number, such as '33' or '2'.\n\n";
		help();
		exit;
```

```
        }
}
```

First, we assign the result of the shift to a variable—our $limit variable. The next item on the command line after --limit should be the number of lines we want to limit to. Just to make sure, we check:

```
if ($limit !~ /^[1-9][0-9]*$/) {
```

This is a different form of regular expression. Instead of "=~" it is "!~". This will match if the regular expression *doesn't* match the text on the left. Remember, an exclamation point often stands in for the word *not* in Perl. Other than that, this regular expression is the same as any other.

We've already met the caret. When it is at the beginning of a regular expression, it matches the beginning of the text. The dollar sign, when it appears at the end of a regular expression, matches the *end* of the text. Square brackets match a *list* or *range* of characters. Here we're using them to match a range. The first character must be a digit from 1 to 9. The second character must be a digit from 0 to 9. Normally, this would mean that the *limit* would have to be 10 to 19. However, immediately following the "[0-9]" there is an asterisk. The asterisk is just like the plus symbol in regular expressions except that it matches *zero* or more occurrences of the preceding piece of text instead of *one* or more.

Since the preceding character is any digit from 0 to 9, the combination of "[0-9]" and an asterisk matches zero or more digits. Matches would include 100, 1, 9, 19, 55, 637. Non-matches would include 01, 99X, Buffalo99. Any text that includes non-numbers or that begins with a zero will not match.

If the text following the --limit switch isn't a number, the script warns them that it needs a number there, calls the *help* subroutine, and exits.

So, that's a little bit more complicated of a switch. How do we handle implementing it?

Replace the line that prints the song information with:

```
#print the information if this line is one we want
if ($matched) {
        $matches++;
        print "$song ($album, by $artist)\n";
}
last if $limit && $matches >= $limit;
```

We've moved the *if* off of the *print* line and instead created an *if* block. We have Perl perform this block if $matched has something in it, that is, if it is *true*. If we have a match, the first thing we do is *increment* the variable $matches by 1. That's what the "++" does. When "++" follows a variable, Perl will add one to that variable. If the variable doesn't exist, or if it is not a number, Perl assumes it is 0 and sets it to 1.

Thus, $matches will count up the number of matches we have hit so far.

Outside of the *if* block, we have a new command: *last*. The *last* command exits the current loop, even if the loop wouldn't otherwise be finished.

We have an *if* following the *last* command, however, so the *last* only gets performed if "$limit && $matches >= $limit".

In other words, if $limit has something in it AND if $matches is greater than or equal to $limit.

If $limit doesn't have anything in it—if we didn't specify a limit—the *last* never gets performed. If $limit *does* have something in it, the *last* will get performed if $matches ever equals or exceeds the limit we specified on the command line.

Remember to add the help line:

```
print "\t--limit x: limit to x results\n";
```

You can now do searches and limit the results. Try the non-best-of search again, with a limit of 10:

```
./show --reverse --limit 10 best songs.txt
./show --reverse --limit 12 aerosmith songs.txt
```

And the screen no longer fills up with the thousands of non-matching songs.

Multiple options

Some switches will have a small list of options to choose from. For example, we modified our script to display the song information in a more human-readable format. But what if we want to keep the raw format under some circumstances? Maybe we want to take the raw song listing, filter out some albums we no longer have, and then create a new raw listing from that filter.

It we want that, it makes sense to create a --format switch that can take only two options: raw and (say) simple. Add the following lines to the switch section:

```
} elsif ($switch eq "format") {
	$format = shift;
	if ($format ne "raw" && $format ne "simple") {
		print "\nFormat must be raw or simple.\n\n";
		help();
		exit;
	}
}
```

Pretty normal stuff here. The letters "ne" stand for "not equal" to. So, if the user specifies a format that is not raw *and* that is not simple, the script displays the help and exits.

Now, replace the line that prints the song information with:

```
if ($format eq "raw") {
	print;
} else {
	print "$song ($album, by $artist)\n";
}
```

What we're really doing here is printing the current line if the user specified a format of *raw*, and printing the simpler information in every other case. But this may change later if we add more

formats. You will usually want to include the default as an option, just in case you make changes later.

```
./show --limit 12 --format raw aerosmith songs.txt
./show --limit 12 --format simple aerosmith songs.txt
```

If you were filtering out information to a new file, you might do this to redirect the output to that file:

```
./show --format raw aerosmith songs.txt > aerosmith.txt
./show love aerosmith.txt
```

You can also type “more aerosmith.txt” to verify that it has what you expect: all songs by Aerosmith.

You’ll want to add format to the help subroutine:

```
print "\t--format <raw or simple >: choose format for results\n";
```

Script confusion

What happens if you misspell a switch? Try:

```
./show --limitt 12 aerosmith songs.txt
```

What is it doing? Those lines don’t contain “aerosmith”. It’s understandable that the script wouldn’t stop at 12 because we misspelled limit, but what is it showing us? Try:

```
./show --limitt 12 aerosmith songs.txt | more
```

and then scroll up a line:

```
Can't open aerosmith: No such file or directory at ./show line 32.
```

It isn’t looking for lines containing “aerosmith”. It thinks “aerosmith” is a file that it needs to search through. What is it looking for? All lines that contain the mention of “12”. That’s because the script saw --limitt as a possible switch, and shifted it off the argument list. But it did not see 12 as a possible switch so it left it on. Our script grabs the first item on the argument list as what to search for. In this case, that was 12.

What we need to do is have the script stop when it hits something it doesn’t understand. That’s easy enough to do. Add another switch:

```
} else {
	print "\nI do not understand the option '$switch'.\n\n";
	help();
	exit;
}
```

This section must always be the *final* section of the switch area. If we’re in the switch area it is because the script saw a double-dash. If we get to the final “else”, that is because none of our known switches matched the text following the double-dash. That’s going to be either because the user misspelled it or because the user doesn’t understand what this script does.

So, we have the script tell the user this, call the *help* subroutine, and exit.

The current script

Just so we're on the same page, here is what the script currently looks like:

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre
#strip off the command-line switches and act on or remember them
while ($ARGV[0] =~ /^--(.+)/) {
	$switch = $1;

	#pull this switch off of the front of the list
	shift;

	#if they ask for help, do it and exit
	if ($switch eq "help") {
		help();
		exit;
	} elsif ($switch eq "case") {
		$sensitive = 1;
	} elsif ($switch eq "reverse") {
		$reverse = 1;
	} elsif ($switch eq "limit") {
		$limit = shift;
		if ($limit !~ /^[1-9][0-9]*$/) {
			print "\nYou must limit to a number, such as '33' or '2'.\n\n";
			help();
			exit;
		}
	} elsif ($switch eq "format") {
		$format = shift;
		if ($format ne "raw" && $format ne "simple") {
			print "\nFormat must be raw or simple.\n\n";
			help();
			exit;
		}
	} else {
		print "\nI do not understand the option '$switch'.\n\n";
		help();
		exit;
	}
}
#the first item on the command line is what we're searching for
if ($searchFor = shift) {
	while (<>) {
		#split out the song, duration, artist, and album
		($song, $duration, $artist, $album) = split("\t");

		if ($sensitive) {
			$matched = /$searchFor/;
		} else {
			$matched = /$searchFor/i;
		}

		#reverse the match if we want non-matching lines
		if ($reverse) {
			$matched = !$matched;
		}

		#print the information if this line is one we want
		if ($matched) {
			$matches++;
			if ($format eq "raw") {
				print;
			} else {
				print "$song ($album, by $artist)\n";
			}
		}
```

```
				last if $limit && $matches >= $limit;
		}
} else {
		help();
}
#describe how this script is used
sub help {
		print "Syntax: show <search text> [song files]\n";
		print "\tSearch for <search text> in the song file. If no song file is specified\n";
		print "\t'show' will expect it on standard input.\n";
		print "\tA song file is a tab-delimited file with:\n";
		print "\ttitle, duration, artist, album, year, rating, rip date, track position, genre\n";
		print "\t--help: print this help text\n";
		print "\t--case: be sensitive to upper and lower case\n";
		print "\t--reverse: filter out songs that contain the search text\n";
		print "\t--limit x: limit to x results\n";
		print "\t--format <raw or simple >: choose format for results\n";
}
```

Arrays and functions

We've done a little bit with an array already: the list of arguments to the script is a simple array. We've only ever referenced the first item in that array, shifting that first item out so that the next items is now first. We can do quite a bit more with arrays in Perl.

Besides simple arrays, there are also *associative* arrays. An associative array is one which, instead of using numbers to reference the values in the array, uses keys. It associates a key with a value. So instead of asking for the first, second, or third item in the list, you can ask for the value that corresponds to "The Band", or the value that corresponds to "Jane Jensen".

For example, we might want to create a third format, one that summarizes songs by artist, showing how many songs each artist has in the matches.

If we're going to have a bunch of formats, it will be easier to keep a list of them. Add the following lines just above the "strip off the command-line switches" section:

```
#options for the --format switch
@validFormats = ("raw", "simple", "summary");
$validFormats = join(", ", @validFormats);
```

The first line (below the comment) assigns a simple array of three items: raw, simple, and summary. I mentioned it in passing earlier, but all simple arrays begin with the @ symbol.

The second line assigns the result of the "join" function to a scalar variable called $validFormats. The "join" function combines an array into a scalar, using the first argument as its glue. Here, we specify a command and a space as the "glue", so $validFormats will be "raw, simple, summary".

Functions are like subroutines, but they are built in to Perl.

Don't get confused by the fact that the scalar variable $validFormats and the simple array @validFormats have the same text for their name. They are not the same variable, and as far as Perl is concerned they are completely unrelated.

Now, inside the switches area, change, the "if ($format ne…" line and the print following it to:

```
if (!grep(/^$format$/, @validFormats)) {
        print "\nFormat must be $validFormats.\n\n";
```

The second line is simple enough: instead of us typing the valid formats, we're using the automatically-created variable that holds them as a piece of text.

The first line uses the *grep* function to check whether or not $format exists in the array @validFormats. Like *join*, grep takes two arguments, and the second one is a list. The first one, however, is a regular expression. So in that line, grep is checking to see if any of the items in @validFormats begins and ends with $format: the caret anchors $format to the beginning, and the dollar sign anchors it to the end.

Go ahead and try a few options and see how they work. Both ‘simple’ and ‘summary’ will currently do the same thing, since we haven’t added any code for ‘summary’.

```
./show --format unknown girl aerosmith.txt
./show --format raw girl aerosmith.txt
./show --format summary girl aerosmith.txt
```

So the next step is to handle the *summary* format. Where the script prints out the song information, between the raw and simple format, add:

```
} elsif ($format eq "summary") {
        $artists{$artist}++;
```

That section should now be:

```
if ($format eq "raw") {
        print;
} elsif ($format eq "summary") {
        $artists{$artist}++;
} else {
        print "$song ($album, by $artist)\n";
}
```

Go ahead and try for a summary:

./show --format summary girl songs.txt

Nothing should happen. When we ask for a summary, we are no longer printing anything, but only keeping track by incrementing… what are we incrementing?

```
$artists{$artist}++;
```

The “++” we’ve already met: it increments the variable to the left of it. The variable to the left looks vaguely like a value from a simple array, except that instead of using square brackets we’re using curly brackets. That’s how you tell the difference between a simple array and an associative array. Simple arrays use square brackets to get at their individual values, and associative arrays use curly brackets to get at their individual values.

If $artist contains “Eurythmics”, this will add one to the value of $artists{"Eurythmics"}. If that value didn’t previously exist, it is assumed to be 0 and now is 1. If it was 1, it is now 2, and so on.

Finally, just outside of the end of the *while* block that loops through the song information, we can print out the summary:

```
if (%artists) {
        @artists = keys %artists;
        @artists = sort @artists;
        foreach $artist (@artists) {
                $artistCount = $artists{$artist};
                print "$artist: $artistCount\n";
        }
}
```

If the associative array %artists exists—that is, if we’ve been keeping track of how many songs each artist has—we’ll perform the rest of this *if* block.

The first line inside the block gets the keys out of the %artists associative array. The keys are a simple list, so they go into @artists.

The second line sorts @artists, and then assigns the sorted @artists back to itself.

The next block is a *foreach* block. Very much like a *while* block, it loops through its lines for as long as it has something to loop through. The difference is that *foreach* gets its things to loop through from a simple array, in this case @artists. *Foreach* places each piece into the first item, in this case the scalar variable $artist.

So if there are three matching artists, Pink Floyd, Warren Zevon, and Stillwater, the first time through $artist will contain "Pink Floyd", the second time through "Warren Zevon", and the third time through "Stillwater".

Inside the *foreach* block, the first line assigns the artist's total songs to the variable $artistCount, and the second line prints out the artist's name and count.

```
./show --format summary stand songs.txt
```

You should get several lines, including that Bing Crosby has 23 songs, Taco 11, and William S. Burroughs 1 matching "stand".

Change the help subroutine to reflect the new format:

```
print "\t--format <$validFormats>: choose format for results\n";
```

Sort numerically

By default, "sort" will sort alphabetically by value. But if we're willing to write our own subroutine we can sort by pretty much any criteria we want. Create a "byArtistCount" subroutine:

```
sub byArtistCount {
	return $artists{$b} <=> $artists{$a};
}
```

Add this to the sort line:

```
@artists = sort byArtistCount @artists;
```

And run the command again:

```
./show --format summary stand songs.txt
```

The top four artists should be Judy Garland, Bing Crosby, The Lennon Sisters, and Linda Ronstadt, at 51 songs, 23 songs, 14 songs, and 12 songs, respectively.

This subroutine is a special one for sorts. When *sort* calls a sort subroutine it is asking that subroutine which of two items should come first. Perl automatically puts the first item in $a and the second item in $b. If the subroutine returns a negative 1, *sort* assumes that $a comes first. If the subroutine returns a positive 1, *sort* assumes that $b comes first. If the subroutine returns a 0, *sort* assumes that both can be ordered either way.

The “<=>”” is a useful operator for sort subroutines, because it returns a negative 1 if the number on the left is lower, a positive 1 if the number on the right is lower, and a zero if both numbers are the same.

Which means that this subroutine ends up sorting the artist names according to their count in %artists; and because I’ve put $b on the left and $a on the right, it sorts in descending order.

If you’re comparing two pieces of text rather than two numbers, the “cmp” operator does the same thing for text that “<=>” does for numbers. Here’s a quick script that lets you play around with compares:

```
#!/usr/bin/perl

$item1 = shift;
$item2 = shift;

print "Text compare: ", $item1 cmp $item2, "\n";
print "Number compare: ", $item1 <=> $item2, "\n";
```

Call it “compare”, make sure you set it to chmod u+x, and play around with giving it two items:

```
./compare hello world
./compare 3 5
```

A smarter join

Go back and ask for some format that doesn’t exist:

```
./show --format wriggling stand songs.txt
Format must be raw, simple, summary.
```

That should really be raw, simple, *or* summary. It’s grating to read otherwise. We can make our own subroutine that joins lists together but accepts a conjunction as well as a simple separator.

```
sub englishJoin {
	my($punctuation) = shift;
	my($conjunction) = shift;
	my(@items) = @_;

	my($joined, $finalItem);

	if ($#items == -1) {
		$joined = "";
	} elsif ($#items == 0) {
		$joined = $items[0];
	} elsif ($#items == 1) {
		$joined = "$items[0] $conjunction $items[1]";
	} else {
		$finalItem = pop(@items);
		$joined = join($punctuation, @items) . "$punctuation$conjunction
$finalItem";
	}
```

```
		return $joined;
	}
```

This subroutine is expecting that the first parameter it gets is the punctuation (the comma, in our case), the second item it gets is the conjunction (“or”), and the rest of the items is the list that needs to be joined. The symbols @_ in a subroutine mean the list of parameter the subroutine has received, much like @ARGV means the list of command-line arguments. Inside of a subroutine, *shift* automatically shifts items out of @_ instead of @ARGV.

Subroutines, by default, have access to all of the variables that the script uses. We used this to our advantage in the byArtistCount sort script. However, most of the time we want to make sure that the variables we use in a subroutine don’t accidentally clobber the other variables used in the script.

Any variable inside of a my() is “local” to the current subroutine. If another variable outside of the subroutine has the same name, that other variable won’t affect the “my” variable, and the “my” variable won’t affect that wider variable.

It is *always* a good idea to automatically “my” any variables a subroutine uses, unless you specifically want to be referencing outside variables.

The characters “$#” in front of a variable name count up the number of items in that array. More specifically, it gives you the current highest item in that simple array. If the array currently has three items in it, the current highest item number is 2, and that’s what “$#” will give you. If the array has one item in it, the current highest item number is 0, and that’s what “$#” will give you.

So we have different *if* blocks depending on whether there are no items in the list (negative one), one item, two items, or three or more items.

Instead of using “eq” to check what $#items is equal to, we are using two equal signs. Perl uses “eq” and “ne” for comparing *text*. It uses “==” and “!=” for comparing *numbers*. This is important because Perl doesn’t care whether a variable is text or is a number until you ask it to make the comparison. Go back to your “compare” script and type:

```
./compare 10 2
```

You should get:

```
Text compare: -1
Number compare: 1
```

Alphabetically, 10 comes before 2. Numerically, 2 comes before 10. With a text compare “10.0” will not equal “10”. But numerically, 10.0 will equal 10. Use the correct operator depending on whether you want to compare as text or compare as a number.

Here, we are comparing as numbers.

The final “else” has a few new things in it also. The *pop* function is the same as *shift* except that it takes an item off of the *end* of the array instead of the beginning.

Those are periods between the “join(…)” function and the text in quotes. If you want to add two numbers together, you use “+”. But if you want to add two strings to each other you use a period. This is also sometimes called *concatenation*.

Change

```
$validFormats = join(", ", @validFormats);
```

to

```
$validFormats = englishJoin(", ", "or", @validFormats);
```

And now:

```
./show --format wriggling stand songs.txt
Format must be raw, simple, or summary.
```

So, now it works, and it will work for any future formats that we add. We also have a new subroutine available if we need a more readable join for any list.

Format conversions

It is now very easy to add new formats. One common use of Perl is to convert data into HTML. Our song listings could just as easily be turned into HTML table rows for insertion into an HTML table.

First, add a new format called "html" to @validFormats.

```
@validFormats = ("raw", "simple", "html", "summary");
```

Second, add a new "elsif" to the part of the script that displays the data:

```
} elsif ($format eq "html") {
        print "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
```

Now, repeat some of your previous searches, but ask for the format to be html instead. The data will be displayed in rows that could be included as part of a web page:

```
./show --album yellow --song girl --format html songs.txt
<tr><td>Young Girl Blues</td><td>Mellow Yellow</td><td>Donovan</td></tr>
<tr><td>Dirty little girl</td><td>Goodbye Yellow Brick Road</td><td>Elton John</td></tr>
<tr><td>All the girls love Alice</td><td>Goodbye Yellow Brick Road</td><td>Elton John</
td></tr>
```

If your web server supports server-side includes, you can automatically include this in your web page. Write it to a file using ">" redirection and include that file.

The current script

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre

#options for the --format switch
@validFormats = ("raw", "simple", "html", "summary");
$validFormats = englishJoin(", ", "or", @validFormats);
#strip off the command-line switches and act on or remember them
while ($ARGV[0] =~ /^--(.+)/) {
        $switch = $1;
```

```
		#pull this switch off of the front of the list
		shift;
		#if they ask for help, do it and exit
		if ($switch eq "help") {
			help();
			exit;
		} elsif ($switch eq "case") {
			$sensitive = 1;
		} elsif ($switch eq "reverse") {
			$reverse = 1;
		} elsif ($switch eq "limit") {
			$limit = shift;
			if ($limit !~ /^[1-9][0-9]*$/) {
				print "\nYou must limit to a number, such as '33' or '2'.\n\n";
				help();
				exit;
			}
		} elsif ($switch eq "format") {
			$format = shift;
			if (!grep(/^$format$/, @validFormats)) {
				print "\nFormat must be $validFormats.\n\n";
				help();
				exit;
			}
		} else {
			print "\nI do not understand the option '$switch'.\n\n";
			help();
			exit;
		}
}
#the first item on the command line is what we're searching for
if ($searchFor = shift) {
		while (<>) {
			#split out the song, duration, artist, and album
			($song, $duration, $artist, $album) = split("\t");

			if ($sensitive) {
				$matched = /$searchFor/;
			} else {
				$matched = /$searchFor/i;
			}
			#reverse the match if we want non-matching lines
			if ($reverse) {
				$matched = !$matched;
			}
			#print the information if this line is one we want
			if ($matched) {
				$matches++;
				if ($format eq "raw") {
					print;
				} elsif ($format eq "html") {
					print "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
				} elsif ($format eq "summary") {
					$artists{$artist}++;
				} else {
					print "$song ($album, by $artist)\n";
				}
			}
			last if $limit && $matches >= $limit;
		}
		if (%artists) {
			@artists = keys %artists;
			@artists = sort byArtistCount @artists;
			foreach $artist (@artists) {
				$artistCount = $artists{$artist};
				print "$artist: $artistCount\n";
			}
		}
} else {
		help();
}
```

```
#describe how this script is used
sub help {
	print "Syntax: show <search text> [song files]\n";
	print "\tSearch for <search text> in the song file. If no song file is specified\n";
	print "\t'show' will expect it on standard input.\n";
	print "\tA song file is a tab-delimited file with:\n";
	print "\ttitle, duration, artist, album, year, rating, rip date, track position, genre\n";
	print "\t--help: print this help text\n";
	print "\t--case: be sensitive to upper and lower case\n";
	print "\t--reverse: filter out songs that contain the search text\n";
	print "\t--limit x: limit to x results\n";
	print "\t--format <$validFormats>: choose format for results\n";
}
sub byArtistCount {
	return $artists{$b} <=> $artists{$a};
}
sub englishJoin {
	my($punctuation) = shift;
	my($conjunction) = shift;
	my(@items) = @_;
	my($joined, $finalItem);
	if ($#items == -1) {
		$joined = "";
	} elsif ($#items == 0) {
		$joined = $items[0];
	} elsif ($#items == 1) {
		$joined = join(" $conjunction ", @items);
	} else {
		$finalItem = pop(@items);
		$joined = join($punctuation, @items) . "$punctuation$conjunction $finalItem";
	}
	return $joined;
}
```

Custom search

Remember that search for “stand” that topped the list with a bunch of older artists? Try that search again without asking for a summary and most of them don’t have “stand” anywhere in the song, artist, or album.

The genre for those songs is *Standards*. Ask for raw format and you’ll see that. Our search is searching through the entire line, both the stuff we can see and the stuff we can’t.

Currently, our script searches everything for the text we specify. It would be nice to be able to focus our search on just the artist, just the album, or just the song. This way, we can search for songs about Yellow without songs about Yellow without picking up albums that mention Yellow.

If we want to search for all songs that mention “yellow” by an artist whose name contains “joni”, we might use:

```
./show --artist Joni --song yellow songs.txt
```

The first step to doing this is to add artist, song, and album to the list of switches.

First, make a list of valid fields to search in:

```
#options for fields to search in
@validFields = ("artist", "album", "song");
$validFields = englishJoin(", ", "and", @validFields);
```

Second, add another *elif* to the switches area:

```
		} elsif (grep(/^$switch$/, @validFields)) {
			if ($searchText = shift) {
				$searches{$switch} = $searchText;
			} else {
				print "\nSearching in $switch requires text to search on.\n\n";
				help();
				exit;
			}
```

We’re storing the search text in an associative array whose key is the field we want to search on.

Because we are now going to be doing multiple searches, we’re going to want a subroutine to do the search. Otherwise, we’ll have to duplicate the “if ($sensitive)” lines for each field we want to search on:

```
sub match {
	my($searchIn) = shift;
	my($searchFor) = shift;
	my($matched) = 0;

	if ($sensitive) {
		$matched = $searchIn =~ /$searchFor/;
	} else {
```

```
                $matched = $searchIn =~ /$searchFor/i;
            }

            return $matched;
    }
```

Change “if ($searchFor = shift) {“ to:

```
    if (%searches) {
```

Instead of expecting some search text, we’re now checking to see if at least one of the searches has been specified. The *if* block will only be performed if the associative array called “searches” exists and isn’t empty.

And finally, replace the “if ($sensitive)” blocks with:

```
    foreach $searchField (keys %searches) {
            $needle = $searches{$searchField};
            $haystack = $$searchField;
            $matched = match($haystack, $needle);
            last if !$matched;
    }
```

Go to the command line and type:

./show --album yellow --song girl songs.txt

You should get back three songs. The albums “Mellow Yellow” and “Goodbye Yellow Brick Road” both contain at least one song whose title contains “yellow”.

First, we assign the number ‘1’ to the variable $matched. By default, we’re assuming that we found a match.

Next, we loop through each field for which we want to search for text. For each such field:

1. We pull the text we’re looking for back out of the “searches” associative array, and assign that text to the variable $needle.

2. We grab the haystack—the text of the current field, that we want to search through, through a little trick called *dereferencing* a *symbolic reference*. Imagine that we are searching for an artist. The %searches array contains “artist” as the key and “some text” as the value. So, $searchField will be “artist”. Now, look up above and see that we have a variable called $artist. If $searchField is “artist”, then $$searchField is the same as $artist. So when we say $haystack = $$searchField, this is the same as saying $haystack = $artist.

3. We set $matched to whether or not $needle can be found in $haystack. If the needle can’t be found, $matched will be false.

4. If $matched is false, there is no need to go any further, so the *last* line exits if !$matched.

5. At the end of this loop, $matched is either true or false. If it is *true*, this track matched our search. Otherwise it did not. It failed at least one of the searches requested on the command line.

If $matched can go through all three checks without becoming zero, that means that this song matches our search. Remember that some checks will be skipped, and thus not affect $matched.

Go ahead and play around with some searches. You can find all of the Elton John songs about girls on albums about yellow, with:

```
./show --album yellow --song girl --artist "Elton John" songs.txt
```

All of the Elton John songs about girls can be found with:

```
./show --song girl --artist "Elton John" songs.txt
```

And, of course, don't forget to add a line to the help for this item! You'll need to change the top item:

```
print "Syntax: show [options] [song files]\n";
```

And add a few lines to the bottom:

```
print "\t--$validFields <searchtext>: search in the $validFields field\n";
print "At least one of the search requests must be specified.\n";
```

That's it!

Symbolic references can be taken to any level. If $key contains "artist", $artist contains "Baez", and $Baez contains "Joan", then $$key is the same as $artist which is the same as "Baez". $$$key is the same as $$artist which is the same as $Baez which is the same as "Joan". Symbolic references are a powerful tool, but can easily make your script confusing. Use them carefully.

New fields

Now that we have an array of valid fields to search through, it's easy enough to add new ones. Go ahead and add "genre" to the list of valid fields:

```
@validFields = ("artist", "album", "song", "genre");
```

At the moment, the script doesn't know about genre, so let's tell the script about all of the fields the file has. Change the "split" lines to:

```
#split out the song information
($song, $duration, $artist, $album, $year, $rating, $ripdate, $track, $genre) = split("\t");
```

That's it. Our script can now limit searches on genre as well as on artist, album, or song:

```
./show --artist linda --genre standard songs.txt
./show --genre video songs.txt
./show --genre spoken --song senator songs.txt
```

Custom sort

If you don't get dereferencing, go back and take another look at it, because we're going to do a different kind of dereferencing here. Arrays can have multiple dimensions. So far, all of the arrays we've used have had a single dimension: our simple arrays have been a list of single items, and our

associative arrays have been simple sets of keys and values. But arrays can have rows and columns much like a spreadsheet; they can even mix simple arrays in one column with associative arrays in others.

Adding a sort switch is pretty easy. We'll want to be able to sort on any valid field, so we can re-use @validFields for this purpose.

```
} elsif ($switch eq "sort") {
        $sortby = shift;
        if (!grep(/^$sortby$/, @validFields)) {
                print "\nI can only sort by $validFields.\n\n";
                help();
                exit;
        }
} else {
```

Because we want to sort the results, we can't just print out each line as soon as we reach it. We'll need to save it for later. Replace the print for raw format with:

```
$text = $_;
```

Replace the print for html format with:

```
$text = "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
```

Replace the print for simple format with:

```
$text = "$song ($album, by $artist)\n";
```

As a test, you might run the script now; you should see nothing, because we aren't printing anything anymore.

After the section the sets the text (and the used to print the text) add:

```
#store or print the display text and the sort text
if ($sortby) {
        $matches[$#matches+1]{'text'} = $text;
        $matches[$#matches]{'sort'} = $$sortby;
} else {
        print $text;
}
```

So, if $sortby exists and has something in it we store the $text we just set for later sorting. If we aren't going to be sorting it, we just print it out now. The interesting part is how we remember this text. We have to remember not only the text we want to display, but also the text we want to sort on.

```
$matches[$#matches+1]{'text'} = $text;
$matches[$#matches]{'sort'} = $$sortby;
```

The first line remembers the text. We're setting up an @matches array that will contain this information. This will be a *simple* array: it will simply be a list of items that goes from 0 on up to however many we find. For a simple array, recall that $#arrayname is the current top item. This means that $#arrayname+1 is the next empty item. That's what we're setting right here: the next empty item in @matches is getting a new item.

That new item is, rather than a scalar variable, an associative array. The first association in that array will be between the word “text” and the display text we want to remember.

The second line remembers what we’re sorting by. Here, we only use $#matches, because the topmost item is the one we want: the previous line added a new item to @matches, and we want to add a new association to the associative array we put there.

We associate the word “sort” with the value of the field we want to sort by. This, again, is a symbolic dereference. If $sortby contains “genre”, $$sortby will be $genre.

So if the first matching song is “Sleeping Bag” by “ZZ Top” from the album “Afterburner”, and we are sorting by song, the first item in @matches ($matches[0]) will be an associative array associating “text” with “Sleeping Bag (Afterburner, by ZZ Top)” and associating “sort” with “Sleeping Bag”.

We’re almost ready to try it. Just to make sure we’re on the same page, here is the entire “if ($matched)” section:

```
if ($matched) {
		$matches++;
		if ($format eq "raw") {
				$text = $_;
		} elsif ($format eq "html") {
				$text = "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
		} elsif ($format eq "summary") {
				$artists{$artist}++;
		} else {
				$text = "$song ($album, by $artist)\n";
		}

		#store or print the display text and the sort text
		if ($sortby) {
				$matches[$#matches+1]{'text'} = $text;
				$matches[$#matches]{'sort'} = $$sortby;
		} else {
				print $text;
		}
}
```

All that’s left is to sort and display the matches. But in order to sort the matches, we need a subroutine that we can hand to *sort*, that knows how to sort matches.

```
sub byCustom {
		return $$a{'sort'} cmp $$b{'sort'};
}
```

When *sort* calls a subroutine, it does not pass arrays. If the item it is passing is an array, it passes a *hard reference* to an array. Just as with symbolic references, we need to dereference a hard reference in order to get at its value.

Here, $a and $b are going to be hard references to an associative array, because each item in @matches is an associative array, and we want to sort @matches. Since we want to sort on the text that is associated with the word ‘sort’ in the associative array, we dereference each array and then ask for the value associated with “sort” in that array.

Remember that "cmp" is the text equivalent of "<=>".

We can also dereference such a reference and get an associative array back by using %$a or %$b. For example, "%leftside = %$a" would make %leftside be a normal associative array that we could get keys from or pull values from as normal.

So, now we have our sort routine. We can finally sort and display our matches.

We already have a place outside of the while that is displaying stored information: there's an "if (%artists)" block. At the end of that block, and an "elsif" block:

```
} elsif (@matches) {
	@matches = sort byCustom @matches;
	foreach $match (@matches) {
		print $$match{'text'};
	}
}
```

We sort @matches, assign the sorted array back to @matches, and then go through @matches for each item it contains. Arrays in Perl do not really contain arrays. They contain hard references to arrays. So we have to dereference that hard reference in order to get the value associated with "text" that we want to display.

```
./show --song shoes songs.txt
./show --song shoes --sort song songs.txt
./show --song shoes --sort artist songs.txt
```

The first one should show about ten songs that mention "shoes" in the title. The second one should show the same songs, but sorted by song title. The third shows the same songs sorted by artist name.

Try this:

```
./show --artist "Elton John" --sort song songs.txt
```

Looks like we're not quite done yet. This is sorted by song, but it's putting the upper-case songs first, and the lower-case songs second. First it sorts through A to Z and then a to z.

This is easy enough to fix. We need to make the comparison in the byCustom subroutine not care about upper or lower case. The easiest way to do this is to make the text be all lower case (or all upper case). There is a function for this: lc("text") will convert that text to all lower case. Change the byCustom subroutine to:

```
sub byCustom {
	if ($sensitive) {
		return $$a{'sort'} cmp $$b{'sort'};
	} else {
		return lc($$a{'sort'}) cmp lc($$b{'sort'});
	}
}
```

Now, by default sorts will not care about case, but if we specify –case sorts will be case sensitive:

```
./show --artist "Elton John" --sort song songs.txt
./show --artist "Elton John" --sort song –case songs.txt
```

And add this to the help:

```
print "\t--sort <$validFields>: sort by specified field\n";
```

Backquoting special characters

Go ahead and look up songs from the album “4”:

```
./show --album 4 songs.txt
```

You’ll end up getting about 100 songs from all albums that include the number “4” in the album title. Currently, our searches look for the search text anywhere in the album name, song title, or artist name. What if we specifically want only the albums with that exact name? Let’s add a switch called “exact”:

```
} elsif ($switch eq "exact") {
        $exact = 1;
```

We can implement this immediately after “if (%switches) {“:

#the first item on the command line is what we're searching for

```
#if we're looking for exact matches, set them up ahead of time
if ($exact) {
        foreach $search (keys %searches) {
                $searchText = $searches{$search};
                $searches{$search} = "^$searchText\$";
        }
}
```

We done all of this before except for the “\$”. It just goes through the keys of %searches, and adds “^” to the beginning of the search text and “$” to the end. But within Perl texts the dollar sign means something special. It means replace this with the variable whose name follows. It doesn’t matter that the variable that follows doesn’t exist, because Perl brings variables into existence the moment they’re used.

So, what we do is “backquote” the dollar sign. A backquote in front of a special character tells Perl not to interpret the special character, but rather to leave it as is. You can even backquote backquotes: "\\n" will not be a new line, it will be a backquote and the letter “n”.

You’ll do the same if you need to put a double-quote inside of double-quoted text. Backquote the “"” character. Instead of ending the text, Perl will insert the “"” into the text at that point:

```
$mobster = "Johnny \"Ratface\" Martin";
```

Don’t forget to add it to the help:

```
print "\t--exact: the search text must match exactly\n";
```

As an exercise, you might consider adding a “beginswith” and an “endswith” switch, to match albums, songs, and artists that begin or end with a specific text.

The current script

This script is beginning to be useful. You should start thinking about the data you work with on a regular basis, and how these techniques could automate what you have to do to this data. Scripts like this can easily be set to run automatically through the use of *cron* or similar tools.

Anyway, here is the script so far:

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre
#options for the --format switch
@validFormats = ("raw", "simple", "html", "summary");
$validFormats = englishJoin(", ", "or", @validFormats);
#options for fields to search in
@validFields = ("artist", "album", "song", "genre");
$validFields = englishJoin(", ", "and", @validFields);
#strip off the command-line switches and act on or remember them
while ($ARGV[0] =~ /^--(.+)/) {
		$switch = $1;
		#pull this switch off of the front of the list
		shift;
		#if they ask for help, do it and exit
		if ($switch eq "help") {
				help();
				exit;
		} elsif ($switch eq "case") {
				$sensitive = 1;
		} elsif ($switch eq "reverse") {
				$reverse = 1;
		} elsif ($switch eq "limit") {
				$limit = shift;
				if ($limit !~ /^[1-9][0-9]*$/) {
						print "\nYou must limit to a number, such as '33' or '2'.\n\n";
						help();
						exit;
				}
		} elsif ($switch eq "format") {
				$format = shift;
				if (!grep(/^$format$/, @validFormats)) {
						print "\nFormat must be $validFormats.\n\n";
						help();
						exit;
				}
		} elsif (grep(/^$switch$/, @validFields)) {
				if ($searchText = shift) {
						$searches{$switch} = $searchText;
				} else {
						print "\nSearching in $switch requires text to search on.\n\n";
						help();
						exit;
				}
		} elsif ($switch eq "sort") {
				$sortby = shift;
				if (!grep(/^$sortby$/, @validFields)) {
						print "\nI can only sort by $validFields.\n\n";
						help();
						exit;
				}
		} elsif ($switch eq "exact") {
				$exact = 1;
		} else {
				print "\nI do not understand the option '$switch'.\n\n";
				help();
				exit;
```

```
		}
}
#the first item on the command line is what we're searching for
if (%searches) {
	#if we're looking for exact matches, set them up ahead of time
	if ($exact) {
		foreach $search (keys %searches) {
			$searchText = $searches{$search};
			$searches{$search} = "^$searchText\$";
		}
	}
	while (<>) {
		#split out the song information
		($song, $duration, $artist, $album, $year, $rating, $ripdate, $track, $genre) = split("\t");
		foreach $searchField (keys %searches) {
			$needle = $searches{$searchField};
			$haystack = $$searchField;
			$matched = match($haystack, $needle);
			last if !$matched;
		}
		#reverse the match if we want non-matching lines
		if ($reverse) {
			$matched = !$matched;
		}
		#print the information if this line is one we want
		if ($matched) {
			$matches++;
			if ($format eq "raw") {
				$text = $_;
			} elsif ($format eq "html") {
				$text = "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
			} elsif ($format eq "summary") {
				$artists{$artist}++;
			} else {
				$text = "$song ($album, by $artist)\n";
			}
			#store or print the display text and the sort text
			if ($sortby) {
				$matches[$#matches+1]{'text'} = $text;
				$matches[$#matches]{'sort'} = $$sortby;
			} else {
				print $text;
			}
		}
		last if $limit && $matches >= $limit;
	}
	if (%artists) {
		@artists = keys %artists;
		@artists = sort byArtistCount @artists;
		foreach $artist (@artists) {
			$artistCount = $artists{$artist};
			print "$artist: $artistCount\n";
		}
	} elsif (@matches) {
		@matches = sort byCustom @matches;
		foreach $match (@matches) {
			print $$match{'text'};
		}
	}
} else {
	help();
}
#describe how this script is used
sub help {
	print "Syntax: show [options] [song files]\n";
	print "\tSearch for some text in the song file. If no song file is specified\n";
	print "\t'show' will expect it on standard input.\n";
	print "\tA song file is a tab-delimited file with:\n";
	print "\ttitle, duration, artist, album, year, rating, rip date, track position, genre\n";
	print "\t--help: print this help text\n";
	print "\t--case: be sensitive to upper and lower case\n";
```

```
		print "\t--reverse: filter out songs that contain the search text\n";
		print "\t--limit x: limit to x results\n";
		print "\t--format <$validFormats>: choose format for results\n";
		print "\t--$validFields <searchtext>: search in the $validFields field\n";
		print "\t--sort <$validFields>: sort by specified field\n";
		print "\t--exact: the search text must match exactly\n";
		print "At least one of the $validFields search requests must be specified.\n";
	}
	sub byArtistCount {
		return $artists{$b} <=> $artists{$a};
	}
	sub englishJoin {
		my($punctuation) = shift;
		my($conjunction) = shift;
		my(@items) = @_;
		my($joined, $finalItem);
		if ($#items == -1) {
			$joined = "";
		} elsif ($#items == 0) {
			$joined = $items[0];
		} elsif ($#items == 1) {
			$joined = "$items[0] $conjunction $items[1]";
		} else {
			$finalItem = pop(@items);
			$joined = join($punctuation, @items) . "$punctuation$conjunction $finalItem";
		}
		return $joined;
	}
	sub match {
		my($searchIn) = shift;
		my($searchFor) = shift;
		my($matched) = 0;
		if ($sensitive) {
			$matched = $searchIn =~ /$searchFor/;
		} else {
			$matched = $searchIn =~ /$searchFor/i;
		}
		return $matched;
	}
	sub byCustom {
		if ($sensitive) {
			return $$a{'sort'} cmp $$b{'sort'};
		} else {
			return lc($$a{'sort'}) cmp lc($$b{'sort'});
		}
	}
```

Creating files

Unix-like operating systems provide an easy means of creating files from any program that has an output. Often, you won't even need to worry about creating files, you'll just redirect to a file and let the operating system handle it for you.

```
./show --exact --artist foreigner --format raw songs.txt > foreigner.txt
```

Because you can pipe directly from one program to another on the command line, you sometimes won't even need to create files to store temporary data. If you want to count up how many songs Foreigner has in songs.txt, you can:

```
./show --exact --artist foreigner songs.txt | wc -l
```

Or, one of my favorites,

```
./show --exact --artist foreigner songs.txt | rev
```

But sometimes we do need to create our own files, and Perl makes this easy. Suppose we wanted to be able to create multiple files, perhaps one for each album, or one for each artist?

We can add a switch for this easily enough.

```
} elsif ($switch eq "export") {
	$exportField = shift;
	if (!grep(/^$exportField$/, @validFields)) {
		print "\nI can only export by $validFields.\n\n";
		help();
		exit;
	}
```

This switch is exactly like our *sort* switch. It accepts a valid field; if the user tries to export by something other than a valid field, the script will warn them and exit.

If the data is being sorted, we are going to have to wait until the end to export the files. So to make it easier, we'll simply always wait until the end to export the files. This lets us re-use some of the code for sorting. Change:

```
if ($sortby) {
	$matches[$#matches+1]{'text'} = $text;
	$matches[$#matches]{'sort'} = $$sortby;
} else {
```

to:

```
if ($sortby || $exportField) {
	$matches[$#matches+1]{'text'} = $text;
	$matches[$#matches]{'sort'} = $$sortby if $sortby;
	$matches[$#matches]{'file'} = $$exportField if $exportField;
} else {
```

The script will now remember the matches if either $sortby or $exportField has something in it. We only store the 'sort' association if $sortby has something in it, and we only store the 'file' association

if $exportField has something in it. If $exportField is “album” and $album is “Head Games”, ‘file’ will associate with “Head Games” for this record.

So now we need to change the code that deals with @matches. Change this:

```
} elsif (@matches) {
	@matches = sort byCustom @matches;
	foreach $match (@matches) {
		print $$match{'text'};
	}
}
```

to:

```
} elsif (@matches) {
	@matches = sort byCustom @matches if $sortby;
	foreach $match (@matches) {
		if ($exportField) {
			$filename = $$match{'file'};
			#open the file if we haven't already
			if (!$files{$filename}) {
				if (!open($files{$filename}, ">$filename")) {
					print "Unable to open $filename: $!\n";
					exit;
				}
			}
			$filehandle = $files{$filename};
			print $filehandle $$match{'text'};
		} else {
			print $$match{'text'};
		}
	}

	#close all open files
	foreach $filehandle (values %files) {
		close($filehandle);
	}
}
```

Note that in the second line we now only sort if $sortby has something in it. Otherwise, there’s nothing to sort on.

We’ve added a new section for “if ($exportField)”, so that if $exportField has something in it we will print to a file instead of to the “standard output” (usually the screen).

Before writing to a file, the file has to be “opened”. We need to get a “handle” on the file. Since we need to have a number of files opened it makes sense to store the file handles in an array. This script stores them in an associative array called %files, associating them with the filename.

Before opening the file with that filename, the script checks to see if there is already a handle associated with that filename in %files. The script only opens the file if there is *not* an existing handle associated with that filename.

If a file needs to be opened, the script opens it within an *if*, so that if there's an error opening the file it can print an error and exit. Perl always stores the most recent error in a special variable called "$!". So, if there's a problem opening $filename, we have the script print "Unable to open $filename" and then "$!". The error message is often very useful. For example, if you don't have permission to open a file, the error message will say this.

The important new part is "open($files{$filename}, ">$filename")". The *open* subroutine accepts two parameters. The first is the variable where we want to store the handle to the file. The second is the name of (or path to) the file we want to open. If we want to be able to write to the file, we need to prepend a greater than symbol to the filename. (We can also append to files by prepending *two* greater than symbols to the filename.)

So, if the script can successfully open the file, we now have a handle to it in $files{$filename}. All that remains is to get it (with "$filehandle = $files{$filename}") and print to it.

If you look at some of the previous *print* commands, they have multiple variables or multiple pieces of text, separated by commas. *Print* can accept any number of pieces of text, separated by commas. However, if the first variable is *not* separated by the rest of the variables or text by a comma, *print* assumes that this is a handle to a file, and redirects its output to that file handle.

That's why there is only a space between $filehandle and $$match{'text'} in "print $filehandle $ $match{'text'}".

Finally, after looping through all matches, we grab every value out of %files—each of which is a file handle—and close that file. The phrase "values %files" is the same as "keys %files" except that it gets a simple array of %file's values, rather than a simple array of %file's keys.

Perl will close files for us automatically as soon as the script ends or exits. But I like to close them explicitly as soon as they are no longer needed. Otherwise they hang around, open, until the script ends. Here that's not a big deal but later on we might alter this script and add more functionality at the end. If that functionality involves opening files too, we might run up against the operating system's limit: most operating systems limit the number of files any one program can open.

Having done all of this, we can now grab, say, all albums by foreigner and create a separate file for each one:

```
./show --exact --artist foreigner --export album songs.txt
```

Of course, you're going to want to make sure that no album has the same name as a file you don't want to erase: every time Perl opens a file, it will happily erase an existing file with the same name. We'll see if we can do something about that in the next section.

And, of course, add this to the help subroutine:

```
print "\t--export <$validFields>: export to files named after the specified field\n"
```

You probably don't want to play around too much making export files. It will be very easy to create hundreds of files in your current directory. We'll fix this next.

Creating folders

It's easy enough to change directory when exporting files in order to ensure that the new files go into a specific folder, but if you're using this as part of a cron job it will be easier if you can tell the script which folder you want the export files to go to.

```
	} elsif ($switch eq "folder") {
		if ($exportFolder = shift) {
			if (-e $exportFolder) {
				#if the folder exists, it needs to be a folder
				help("$exportFolder already exists and is not a folder.") if !-d
$exportFolder;
			}
		} else {
			help("The folder option requires a folder name.");
		}
```

What's with the new use of help()? Every single time we use help, we also exit. Every time except one, we print out an error message above the call to help. It's about time we automated this. It can often be difficult to make the decision to change a function when only a minor change is needed; every time we've called help so far, it's been a simple effort to add the extra line above and below the help call. But for simplicity's sake it is time we combined those two to three lines into a single subroutine call.

At the top of the help subroutine, just below "sub help {", add:

```
#if there is an error message, print it out separated from the rest
if (my($message) = shift) {
	print "\n$message\n\n";
}
```

And at the very end, just before "}", add:

```
exit;
```

At some point, you'll want to go through and find every use of:

```
print "some text";
help();
exit;
```

and replace it with:

```
help("some text");
```

If you don't do it now, add a comment to the top of your script reminding you to do it later.

The other part that's new is "-e $exportFolder". There are several tests you can perform on filenames. They all begin with a dash. This is the *exists* test. It is "true" if there is a file with that name. Remember that in Unix, folders are also files.

If that name is already being used, that's fine if it's a folder. So we need to make sure it's a folder. The "-d" file test tests for that. So we call *help* if *not* "-d $exportFolder". If $exportFolder is *not* a directory, the script will print the help, which with the above change will automatically exit the script.

Okay, add this option to our help subroutine:

```
print "\t--folder <foldername>: export files are created in the specified folder\n";
```

Now it's time to implement it. Between "@matches = sort byCustom @matches if $sortby;" and "foreach $match (@matches) {" add:

```
#create a folder if necessary, and move into it
if ($exportField && $exportFolder) {
	if (!-e $exportFolder) {
		if (!mkdir($exportFolder)) {
			print "Unable to create $exportFolder: $!\n";
			exit;
		}
	}
	if (!chdir($exportFolder)) {
		print "Unable to move into $exportFolder: $!\n";
		exit;
	}
}
```

If there is something in $exportField (that is, if we are exporting into some files) and if there is something in $exportFolder (that is, if we are doing this into a specific folder), we need to ensure that the folder exists and that we are in it.

Step one checks to see if $exportFolder already exists, using the -e file test. If it doesn't, the script tries to create it using "mkdir()". This stands for *make directory*. If that works, fine, but if *not* (watch the exclamation point) the script prints the error and exits.

Whether we just created the folder or it already existed, the next step is to get into it. This is the "chdir()" function. It stands for *change directory*. It doesn't mean change *a* directory, but change *into* a directory. It moves the script into that directory so that any files the script creates from now on will be created in that directory.

If chdir is *not* successful, the script prints an error and exits.

We can now export the foreigner albums into their own folder:

```
./show --exact --artist foreigner --export album --folder Foreigner songs.txt
```

This makes it easier for us to export to multiple files without cluttering up the current directory.

Replacing text

If you play around with *export* now, you'll find that some exports don't work. Go ahead and try:

```
./show --artist afroman --export genre songs.txt
```

What you should get back is:

```
Unable to open Hip Hop/Rap
: No such file or directory
```

There are two problems here. One, why is that error message separated onto two lines? It looks like we're printing a new line between the export file name and the colon, but if you look in the code there is no such new line. The second, why would it tell us that there is no such file? We know that: that's why we're trying to create it.

The first problem is an easy one to fix. That new line is in the genre name itself. The genre is the last field on the line. When we get a line of text from a file in Perl, Perl includes the line break at the end of the line. When we *split* the line on tabs, the last item gets this line break, and that last item is the genre.

In front of the *split* line where we get song, duration, artist, etc., add:

```
chomp;
```

This will *chomp* any line endings off of the end of the current line. If you want to chomp the line endings off of a specific variable, you can use "chomp($variable)". If you want to chomp the line endings off of a list of variables, you can use "chomp(@list)" or "chomp($variable1, $variable2, etc.)".

Because we're chomping the line endings off of the current line, we need to change the raw format as well. Change "$text = $_;" to:

```
$text = "$_\n";
```

Now the error message is easier to read:

```
Unable to open Hip Hop/Rap: No such file or directory
```

The problem here is that Unix uses the slash to separate directory names from each other and from the file name. Perl thinks the script wants to create a file called "Rap" in a folder called "Hip Hop". There is no folder called "Hip Hop", so this fails with that error.

We need to get rid of that slash. There is another form of regular expression that does this for us. Let's add a general subroutine for replacing characters in a piece of text:

```
sub replace {
	my($text, $from, $to) = (shift, shift, shift);

	$text =~ s/$from/$to/g;

	return $text;
}
```

Remember that we already know what *shift* does. Here, we're just doing it three times in a row to grab each of the three items we will send this subroutine: the text we want to change, the characters we want to look for in that text, and the characters we want to replace it with. If we call "replace ("omega man", "m", "d")", we would expect to get back "odega dan".

```
$text =~ s/$from/$to/g;
```

The "=~" means that this is a regular expression. The "s" in front of the first slash means that this is a *substitution*. By default regular expressions only *match*, they don't perform any changes. A substitution will. In order to substitute, it needs to know *what* to substitute. That's between the second slash and the new third slash. What we have here as $to will replace $from in $text.

By default, substitutions will only make one substitution. If we want the regular expression to affect *all* occurrences of $from, we need to specify that this is a *global* replace. The "g" after the final slash does this. The "g" is a modifier much like "i" for case insensitive matches.

If we want this to also be case insensitive, we could also add the "i" there too:

```
$text =~ s/$from/$to/gi;
```

What we want is to replace slashes with dashes, so replace "$matches[$#matches]{'file'} = $$exportField if $exportField;" with:

```
if ($exportField) {
        $filename = $$exportField;
        $filename = replace($filename, "/", "-");
        $matches[$#matches]{'file'} = $filename;
}
```

If we run the Afroman export again:

```
./show --artist afroman --export genre songs.txt
```

We now have a file called "Hip Hop-Rap".

You might choose a different character to replace slashes. It must be a valid character for your operating system or you'll continue to get some sort of error.

Try to break it

One of the most important skills to learn when you're programming is learning how to break your scripts. You'll want to do lots of tests with lots of different kinds of data, but tests can only find errors that you test on. You will also need to think about *where will this break?* and fix those errors before they happen. We've done a little of this already, without calling it that. This is why we put "open(…)" in an "if" statement and match it with an "else" that displays any errors that crop up.

It is especially important to think about how new functionality can break your script. So we recently added the ability to search, sort, and export by new fields. We can add any field we want to @validFields and search, sort, and export by that field. How can that break our script?

One way it could break our script is if we try to export on that field but the song doesn't have anything entered for it.

In fact, that might be the case even with our current fields. How can we find fields that don't have anything in them?

We're doing our search by regular expression. We haven't blocked regular expression characters from the search text. Try:

```
./show --artist ^Night songs.txt
./show --artist Night$ songs.txt
```

The first one will show only those songs by artists whose name *begins* with “night”. The second shows artists whose names *end* with “night”. Because that’s the regular expression character for anchoring to the beginning or end of a text.

There’s also a regular expression for *any character*. If we search on that, and then get the reverse, we can find fields that have *no character* in them.

```
./show --artist . --reverse songs.txt
./show --song . --reverse songs.txt
./show --album . --reverse songs.txt
./show --genre . --reverse songs.txt
```

There are artists, albums, and genres that are completely empty. Try to export on artist and you’ll get an error:

```
./show --artist . --reverse --export artist songs.txt
Unable to open : No such file or directory
```

There’s no way to open a file that has no name. We need to check for empty filenames and give them some other name, such as “Unknown artist”.

In the “if ($exportField) area that we just changed, change it again:

```
if ($exportField) {
	$filename = $$exportField;
	if ($filename ne "") {
		$filename = replace($filename, "/", "-");
	} else {
		$filename = "Unknown $exportField";
	}
	$matches[$#matches]{'file'} = $filename;
}
```

Now, *if* $filename ends up being *not equal* to an empty string, we do the replace as normal. Otherwise, we assign “Unknown $exportField” to $filename. If we are exporting by artist, it will say “Unknown artist”.

The “if ($filename ne "") {” is a little different from “if ($filename = $$exportField) {” which we could have done. The former will *only* match if $filename is *empty*. The latter would also match if $filename was zero. We might imagine wanting to export based on rating, so that we have a list of songs each in a file named after their rating. A rating of zero would get a filename of “Unknown rating” if we used the latter form, but will get a filename of “0” with the one we used.

Timestamps

Some data is time-sensitive. The file came in at a specific time, and you want the exported files to keep that timestamp. Under Unix, you can see a file’s last modified time using “ls -l”. If you look at songs.txt you’ll probably see that it was last modified on April 25, 2005. If you look at the export files you’ve created, their last modified time is today, or the day you exported them.

First, add the switch:

```
} elsif ($switch eq "keep-time") {
        $keepTime = 1;
```

and then the help:

```
print "\t--keep-time: keep the input file's timestamp on any exported files\n";
```

If we're going to stamp the files we create so that they have the same timestamp as the file the data came from, we need to get the timestamp of that file. So far we haven't cared what file that is. In fact, our script is designed to allow multiple files to be specified on the command line. We might imagine exporting raw artist files of all Rock songs, for example, and then searching through the files for multiple artists.

So the first step is purely on our part, with no coding. If more than one file is specified, what is the correct timestamp? Do we want the most recent one? The oldest one? Some sort of average? I'm going to assume that we want the most recent one.

The second problem is that in order to get the timestamp for a file, we need to know the file's name. So far we haven't cared. We've let Perl handle the file input for us. Fortunately, there's no need to change that. Perl can also tell us the name of the current file. When a script loops through file input, Perl puts the current filename in a special variable called $ARGV.

Below the "if ($matched) {" line, add:

```
if ($keepTime) {
        @fileInfo = stat($ARGV);
        $fileMod = $fileInfo[9];
        $lastModified = $fileMod if $fileMod > $lastModified;
}
```

Simple enough. If $keepTime has something in it, we grab the information for the file called $ARGV. The stat() function returns a bunch of information about a file; we want the ninth piece. That's the last modified time of the file.

Then, we set $lastModified to be this file's modification time if $fileMod is larger than (more recent than) the current $lastModified. The first time around, $lastModified has nothing in it, so anything will be greater than it. After that, $lastModified only gets changed if the current file is newer than the previous newest file.

One minor problem with this is that it is checking the current file every time we go through the loop. File system access is usually very fast, but if we're exporting thousands of records from a handful of files that's thousands of *stat* calls we don't really need. What we can do is keep track of the filename, and only get the last modified when $ARGV no longer matches the previously current filename:

```
if ($keepTime && $lastFile ne $ARGV) {
        @fileInfo = stat($ARGV);
        $fileMod = $fileInfo[9];
        $lastModified = $fileMod if $fileMod > $lastModified;
        $lastFile = $ARGV;
}
```

So, now we have the timestamp we need, we just need to set each file to have that timestamp. The easiest place to handle this is after we close each file. The script already goes through each file one by

one to close it. We can set the last modified time during that loop. Change the entire "#close all open files" section to:

```
#close all open files
foreach $filename (keys %files) {
	$filehandle = $files{$filename};
	close($filehandle);
	utime($lastModified, time(), $filename);
}
```

Instead of just grabbing the values (file handles) out of %files, we need the keys as well. The keys are the filenames. So, we grab the keys and then grab the values using the key as normal. We close $filehandle just as we always did, and then we run *utime* on $filename. Each file has two times that are commonly used: the last modified time and the last accessed time. The *utime* function requires both of them, so we'll set the first one (the last modified time) to the saved $lastModified from the input file(s). We'll set second (last access time) to the current time, since that's when the file was last accessed.

The current script

This script is getting pretty big, but we're almost done with it. Here is how it stands so far.

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre
#options for the --format switch
@validFormats = ("raw", "simple", "html", "summary");
$validFormats = englishJoin(", ", "or", @validFormats);
#options for fields to search in
@validFields = ("artist", "album", "song", "genre");
$validFields = englishJoin(", ", "and", @validFields);
#strip off the command-line switches and act on or remember them
while ($ARGV[0] =~ /^--(.+)/) {
	$switch = $1;
	#pull this switch off of the front of the list
	shift;
	#if they ask for help, do it and exit
	if ($switch eq "help") {
		help();
	} elsif ($switch eq "case") {
		$sensitive = 1;
	} elsif ($switch eq "reverse") {
		$reverse = 1;
	} elsif ($switch eq "limit") {
		$limit = shift;
		if ($limit !~ /^[1-9][0-9]*$/) {
			help("You must limit to a number, such as '33' or '2'.");
		}
	} elsif ($switch eq "format") {
		$format = shift;
		if (!grep(/^$format$/, @validFormats)) {
			help("Format must be $validFormats.");
		}
	} elsif (grep(/^$switch$/, @validFields)) {
		if ($searchText = shift) {
			$searches{$switch} = $searchText;
		} else {
			help("Searching in $switch requires text to search on.");
		}
	} elsif ($switch eq "sort") {
		$sortby = shift;
```

```
			if (!grep(/^$sortby$/, @validFields)) {
				help("I can only sort by $validFields.");
			}
		} elsif ($switch eq "exact") {
			$exact = 1;
		} elsif ($switch eq "export") {
			$exportField = shift;
			if (!grep(/^$exportField$/, @validFields)) {
				help("I can only export by $validFields.");
			}
		} elsif ($switch eq "folder") {
			if ($exportFolder = shift) {
				if (-e $exportFolder) {
					#if the folder exists, it needs to be a folder
					help("$exportFolder already exists and is not a folder.") if !-d $exportFolder;
				}
			} else {
				help("The folder option requires a folder name.");
			}
		} elsif ($switch eq "keep-time") {
			$keepTime = 1;
		} else {
			help("I do not understand the option '$switch'.");
		}
}
#the first item on the command line is what we're searching for
if (%searches) {
		#if we're looking for exact matches, set them up ahead of time
		if ($exact) {
			foreach $search (keys %searches) {
				$searchText = $searches{$search};
				$searches{$search} = "^$searchText\$";
			}
		}
		while (<>) {
			#split out the song information
			chomp;
			($song, $duration, $artist, $album, $year, $rating, $ripdate, $track, $genre) = split("\t");
			foreach $searchField (keys %searches) {
				$needle = $searches{$searchField};
				$haystack = $$searchField;
				$matched = match($haystack, $needle);
				last if !$matched;
			}
			#reverse the match if we want non-matching lines
			if ($reverse) {
				$matched = !$matched;
			}
			#print the information if this line is one we want
			if ($matched) {
				#maintain the timestamp if we need it
				if ($keepTime && $lastFile ne $ARGV) {
					@fileInfo = stat($ARGV);
					$fileMod = $fileInfo[9];
					$lastModified = $fileMod if $fileMod > $lastModified;
					$lastFile = $ARGV;
				}
				$matches++;
				if ($format eq "raw") {
					$text = "$_\n";
				} elsif ($format eq "html") {
					$text = "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
				} elsif ($format eq "summary") {
					$artists{$artist}++;
				} else {
					$text = "$song ($album, by $artist)\n";
				}
				#store or print the display text and the sort text
				if ($sortby || $exportField) {
					$matches[$#matches+1]{'text'} = $text;
					$matches[$#matches]{'sort'} = $$sortby if $sortby;
```

```
				if ($exportField) {
					$filename = $$exportField;
					if ($filename ne "") {
						$filename = replace($filename, "/", "-");
					} else {
						$filename = "Unknown $exportField";
					}
					$matches[$#matches]{'file'} = $filename;
				}
			} else {
				print $text;
			}
		}
		last if $limit && $matches >= $limit;
	}
	if (%artists) {
		@artists = keys %artists;
		@artists = sort byArtistCount @artists;
		foreach $artist (@artists) {
			$artistCount = $artists{$artist};
			print "$artist: $artistCount\n";
		}
	} elsif (@matches) {
		@matches = sort byCustom @matches if $sortby;
		#create a folder if necessary, and move into it
		if ($exportField && $exportFolder) {
			if (!-e $exportFolder) {
				if (!mkdir($exportFolder)) {
					print "Unable to create $exportFolder: $!\n";
					exit;
				}
			}
			if (!chdir($exportFolder)) {
				print "Unable to move into $exportFolder: $!\n";
				exit;
			}
		}
		foreach $match (@matches) {
			if ($exportField) {
				$filename = $$match{'file'};
				#open the file if we haven't already
				if (!$files{$filename}) {
					if (!open($files{$filename}, ">$filename")) {
						print "Unable to open $filename: $!\n";
						exit;
					}
				}
				$filehandle = $files{$filename};
				print $filehandle $$match{'text'};
			} else {
				print $$match{'text'};
			}
		}
		#close all open files
		foreach $filename (keys %files) {
			$filehandle = $files{$filename};
			close($filehandle);
			utime($lastModified, time(), $filename);
		}
	}
} else {
	help();
}
#describe how this script is used
sub help {
	#if there is an error message, print it out separated from the rest
	if (my($message) = shift) {
		print "\n$message\n\n";
	}
	print "Syntax: show [options] [song files]\n";
	print "\tSearch for some text in the song file. If no song file is specified\n";
```

```
		print "\t'show' will expect it on standard input.\n";
		print "\tA song file is a tab-delimited file with:\n";
		print "\ttitle, duration, artist, album, year, rating, rip date, track position, genre\n";
		print "\t--help: print this help text\n";
		print "\t--case: be sensitive to upper and lower case\n";
		print "\t--reverse: filter out songs that contain the search text\n";
		print "\t--limit x: limit to x results\n";
		print "\t--format <$validFormats>: choose format for results\n";
		print "\t--$validFields <searchtext>: search in the $validFields field\n";
		print "\t--sort <$validFields>: sort by specified field\n";
		print "\t--exact: the search text must match exactly\n";
		print "\t--export <$validFields>: export to files named after the specified field\n";
		print "\t--folder <foldername>: export files are created in the specified folder\n";
		print "\t--keep-time: keep the input file's timestamp on any exported files\n";
		print "At least one of the $validFields search requests must be specified.\n";
		exit;
}
sub byArtistCount {
		return $artists{$b} <=> $artists{$a};
}
sub englishJoin {
		my($punctuation) = shift;
		my($conjunction) = shift;
		my(@items) = @_;
		my($joined, $finalItem);
		if ($#items == -1) {
			$joined = "";
		} elsif ($#items == 0) {
			$joined = $items[0];
		} elsif ($#items == 1) {
			$joined = "$items[0] $conjunction $items[1]";
		} else {
			$finalItem = pop(@items);
			$joined = join($punctuation, @items) . "$punctuation$conjunction $finalItem";
		}
		return $joined;
}
sub match {
		my($searchIn) = shift;
		my($searchFor) = shift;
		my($matched) = 0;
		if ($sensitive) {
			$matched = $searchIn =~ /$searchFor/;
		} else {
			$matched = $searchIn =~ /$searchFor/i;
		}
		return $matched;
}
sub byCustom {
		if ($sensitive) {
			return $$a{'sort'} cmp $$b{'sort'};
		} else {
			return lc($$a{'sort'}) cmp lc($$b{'sort'});
		}
}
sub replace {
		my($text, $from, $to) = (shift, shift, shift);

		$text =~ s/$from/$to/g;
		return $text;
}
```

SQL database

One of the most common things Perl is used for when it comes to raw data files is importing them into databases. To import into databases in Perl, you need a special *module*. This is normally the DBI module, and you'll access it by putting a "use" line at the top of your script. This tells Perl that you want to access an external module for more functionality.

```
use DBI;
$dbh = DBI->connect("dbi:SQLite:dbname=TESTDB");
```

Put this just below your first set of comments, and then run

```
./show
```

If it shows you your help as normal, then you have both DBI and SQLite installed on your system.

If it gives you the error "Can't locate DBI.pm in …", you'll need to install DBI and SQLite.

If it gives you the error "install_driver(SQLite) failed: Can't locate DBD/SQLite.pm in @INC…", this means that you have DBI but you'll need to install SQLite.

Installing from CPAN

Fortunately, Perl has an easy way to install modules. Unfortunately, it isn't so easy if you aren't the system administrator. If you aren't the system administrator, you'll need to ask your system administrator to install it for you. If you are, you can do it yourself.

Your system must have a compiler in order to install these Perl modules. You don't have to know how to use it. Perl will handle that for you. But you will need it. Many Unix-like operating systems come with the compiler pre-installed. If you are using Mac OS X, you'll need to install XCode from your install CD or DVD.

These instructions assume that your account is an administrative one, where "administrative" means that it can use the "sudo" command. Type the following from the command line to enter the Perl shell:

```
sudo perl -MCPAN -e shell
```

The first time you enter CPAN it is going to ask a whole bunch of questions. You can choose the default for all of them. Choose the default by pressing the return key. Note that some questions, such as "Fetching with Net::FTP:" will take a while. It is downloading information from the Comprehensive Perl Archive Network. This is a centrally distributed archive for Perl modules (among other things).

The ones you will have to answer are your continent, your country, and a list of URLs to download from. (Make sure you choose the URLs by number, and separate them by spaces. Just read the directions for that question when it comes up.)

Once cpan is done configuring itself, we can install the modules we need. If you need to install the DBI module, type:

```
install DBI
```

Press return after you type that. If you need SQLite, then, when DBI has finished installing type:

```
install DBD::SQLite
```

Perl will attempt to install what it needs for these modules. It will occasionally ask for permission to download prerequisites. Generally you'll want to say no, but read http://sial.org/howto/perl/life-with-cpan/ for specifics. You shouldn't need anything except DBI and DBD::SQLite for this section of the tutorial.

See http://search.cpan.org/dist/DBI/lib/DBD/DBM.pm for more information about the DBM section of the DBI module for Perl.

Using SQLite

Go ahead and remove that "$dbh = …" line that we added for testing. Leave "use DBI;" there.

What we would like to do is import this data from its file into a database. First, add the switch:

```
} elsif ($switch eq "import") {
	if ($importDB = shift) {
		$importHandle = DBI->connect("dbi:SQLite:dbname=$importDB");
		if (!$importHandle) {
			print "Unable to open database $importDB: $!\n";
		}
	} else {
		help("The import switch requires a database name to import to.");
	}
```

The --import switch will require the name of a database. As soon as we get the database name, we try to open it:

```
$importHandle = DBI->connect("dbi:SQLite:dbname=$importDB");
```

This is a bit of a different format than we've seen before. DBI is an *object*. This is a special programming tool that contains code, called *methods*, that we can access on that object. One of the methods is the *connect* method. It connects to the database. Methods are very much like the subroutines we've already been using. They are almost always accessed using *object->method(parameters)*.

If DBI can't open the database, we exit and try to print an error message.

Add the help:

```
print "\t--import <database name>: import data into a named database\n";
```

You might go ahead and type "./show --import" and make sure that the help is displayed as expected.

Creating tables

Now we know how to open the database. SQL databases require tables. We need to create one if it does not already exist. Let's go ahead and just make a subroutine to do this.

```
#expects a handle to a database connection, a name for a table, and an associative
#array of field names and field types passed as a reference
sub createTable {
	my($dbHandle) = shift;
	my($tableName) = shift;
	my($tableFields) = shift;

	my(@fieldCreators, $createQuery, $queryHandle, $fieldType, $fieldName);

	@tables = $importHandle->tables();
	if (!grep(/^"$dbTable"$/, @tables)) {
		#construct query to create table
		foreach $fieldName (keys %$tableFields) {
			$fieldType = $$tableFields{$fieldName};
			$fieldCreators[$#fieldCreators+1] = "$fieldName $fieldType";
		}
		$createQuery = "CREATE TABLE $tableName (" . join(", ", @fieldCreators) . ")";

		#tell database to create the table
		if ($queryHandle = $dbHandle->prepare($createQuery)) {
			if (!$queryHandle->execute) {
				print "Unable to execute $createQuery: $!\n";
			}
		} else {
			print "Unable to prepare $createQuery: $!\n";
			exit;
		}
	}
}
```

There are two sections to this subroutine. The first one creates the query that will create the table. The second one hands that query off to the database. When you use a SQL query in Perl, you usually hand it off in two steps: first, you *prepare* it on the handle. This creates a "statement handle" for the query statement. Then, you *execute* it from that statement handle. Since both of these handles are objects with methods, we use the *handle->method* syntax to prepare and to execute the query.

Pay close attention to the third parameter that this subroutine is expecting: it's an associative array. We can't pass associative arrays to subroutines in Perl. We can only pass scalar variables and lists of scalar variables. What we'll need to do when we call this subroutine is pass that associative array by *reference*.

If you recall when we were sorting, Perl handed our sort subroutine associative arrays by *hard reference*. Well, we can hand hard references out to our subroutines, too.

Change the "$importHandle=" section of the switch for imports. We'll call the *createTable* subroutine as soon as we open the database.

```
if ($importHandle = DBI->connect("dbi:SQLite:dbname=$importDB")) {
	createTable($importHandle, $dbTable, \%dbFields);
} else {
	print "Unable to open database $importDB: $!\n";
}
```

When you want to pass a variable as a hard reference, precede the variable name with a backslash.

We also need to create the %dbFields associative array. At the top, add the following defaults:

```
#name of table
$dbTable = "music";
#what kind of fields need to be in the database?
%dbFields = (
		"ID"=>"INTEGER PRIMARY KEY AUTOINCREMENT",
		"song"=>"TEXT",
		"duration"=>"TEXT",
		"artist"=>"TEXT",
		"album"=>"TEXT",
		"year"=>"INTEGER",
		"rating"=>"INTEGER",
		"ripdate"=>"TEXT",
		"genre"=>"TEXT"
	);
```

Note that we are putting this "single line" of Perl code onto several lines. Perl only cares about that semicolon. That's how Perl sees the end of a line of code. Separating the associations into separate lines makes it easier for you, as programmer, to read them.

Also, this is the first time we've created an associative array by specifying the keys and values all at once. It is similar to how we create simple arrays, but each item is a key and a value, separated by "=>". The key is on the left and the value is on the right.

Inserting data

Okay, we've opened the database and we've created the table. Now we need to insert data into the table. It is time to add functionality to our --import switch.

First, let's add a subroutine called "insertRow". This subroutine will be similar to the "createTable" subroutine, but it will insert rows of data. We'll pass it the same data: a handle to the database, the name of the table that will receive the data, and the associative array of fieldnames as a hard reference.

This subroutine will also expect that the field names in that associative array match an existing scalar variable with the same name. So if there is a field named "artist", this subroutine will look in $artist for that field's value.

```
#expects a handle to a database connection, a name for a table, and an associative
#array of field names and field types passed as a reference
#also expects that the field names match a currently-existing scalar variable
sub insertRow {
```

```
		my($dbHandle) = shift;
		my($tableName) = shift;
		my($tableFields) = shift;

		my(@fieldNames, @fieldValues, $fieldNames, $fieldValues, $fieldValue);
		my($insertQuery, $queryHandle);

		#create the field list and value list
		@fieldNames = keys %$tableFields;
		foreach $fieldName (@fieldNames) {
			$fieldValue = $$fieldName;
			#set $fieldValue to an appropriate SQL value
			if ($fieldValue eq "") {
				$fieldValue = NULL;
			} else {
				$fieldValue = '$dbHandle->quote($$fieldname);
			}
			$fieldValues[$#fieldValues+1] = $fieldValue;
		}
		$fieldNames = join(", ", @fieldNames);
		$fieldValues = join(", ", @fieldValues);
		$insertQuery = "INSERT INTO $tableName ($fieldNames) VALUES ($fieldValues)";

		#insert into the database
		doQuery($dbHandle, $insertQuery);
	}
```

If you aren't familiar with SQL, this will create queries such as "INSERT INTO music (artist, album, year) VALUES ("Foreigner", "4", NULL);". If a field doesn't have a value—if it equals the empty string ""—then we set the value to NULL. Otherwise, we tell the database to correctly quote this piece of text. We need to do this because SQL requires text to be surrounded by quotes. But some text *contains* quotes. The *quote* method on the database handle understands this, and fixes the text accordingly.

Once we have the query created, executing the query is the same as executing the CREATE TABLE query. Rather than have the same code duplicated for each kind of query, we can make a subroutine to handle it. The subroutine will need the database handle and the query.

```
	sub doQuery {
		my($dbHandle) = shift;
		my($query) = shift;

		if ($queryHandle = $dbHandle->prepare($query)) {
			if (!$queryHandle->execute) {
				print "Unable to execute $query: $!\n";
			}
		} else {
			print "Unable to prepare $query: $!\n";
			exit;
		}
	}
```

You should go replace the "if $queryHandle =" section of the *createTable* subroutine so that it calls

```
#tell database to create the table
doQuery($dbHandle, $createQuery);
```

Now would be a good time to try "./show" with no parameters to see if you have any errors. It should just give you the help message.

Finally, we need to call this *insertRow* subroutine for every matching row. At the end of the "if ($matched) {" section, add:

```
#import into a database?
if ($importDB) {
	insertRow($importHandle, $dbTable, \%dbFields);
}
```

Let's see if we can make a database of love songs:

```
./show --song love --import LoveSongs songs.txt
```

This works, but it is slow. SQLite can be much faster than this. What we need to tell it to do is to wait until we're done feeding it rows before it writes them all out to the disk. After the *createTable* call in our switches area, add:

```
$importHandle->{AutoCommit} = 0;
```

This tells SQLite *not* to automatically write rows out to the disk as soon as it receives them. We'll need to explicitly tell it when to "commit" our changes. Just in front of "if (%artists) {" and just at the end of the loop through all lines, add:

```
#commit insertions if we are importing into a database
$importHandle->commit;
```

Go ahead and remove the file LoveSongs, and run that again:

```
./show --song love --import LoveSongs songs.txt
```

It should go much faster.

How do you know the data has been inserted? If you have sqlite3 on your system, you can type "sqlite3 *databasename*" to open the database by hand. Then, type:

```
SELECT * FROM music;
```

You should see a long list of all of the data that's been imported. Use ".quit" to exit.

If you don't have sqlite3 installed on your system, you'll just have to wait until the next section.

The final script

This is the end of this script. We'll start from scratch on the next one, and display our data.

```
#!/usr/bin/perl
#Search for songs in a file of the following tab-separated data:
# title, duration, artist, album, year, rating, rip date, track position, genre
use DBI;
#options for the --format switch
@validFormats = ("raw", "simple", "html", "summary");
$validFormats = englishJoin(", ", "or", @validFormats);
```

```
#options for fields to search in
@validFields = ("artist", "album", "song", "genre");
$validFields = englishJoin(", ", "and", @validFields);
#database information
#name of table
$dbTable = "music";
#what kind of fields need to be in the database?
%dbFields = (
			"ID"=>"INTEGER PRIMARY KEY AUTOINCREMENT",
			"song"=>"TEXT",
			"duration"=>"TEXT",
			"artist"=>"TEXT",
			"album"=>"TEXT",
			"year"=>"INTEGER",
			"rating"=>"INTEGER",
			"ripdate"=>"TEXT",
			"genre"=>"TEXT"
	);

#strip off the command-line switches and act on or remember them
while ($ARGV[0] =~ /^--(.+)/) {
	$switch = $1;
	#pull this switch off of the front of the list
	shift;
	#if they ask for help, do it and exit
	if ($switch eq "help") {
		help();
	} elsif ($switch eq "case") {
		$sensitive = 1;
	} elsif ($switch eq "reverse") {
		$reverse = 1;
	} elsif ($switch eq "limit") {
		$limit = shift;
		if ($limit !~ /^[1-9][0-9]*$/) {
			help("You must limit to a number, such as '33' or '2'.");
		}
	} elsif ($switch eq "format") {
		$format = shift;
		if (!grep(/^$format$/, @validFormats)) {
			help("Format must be $validFormats.");
		}
	} elsif (grep(/^$switch$/, @validFields)) {
		if ($searchText = shift) {
			$searches{$switch} = $searchText;
		} else {
			help("Searching in $switch requires text to search on.");
		}
	} elsif ($switch eq "sort") {
		$sortby = shift;
		if (!grep(/^$sortby$/, @validFields)) {
			help("I can only sort by $validFields.");
		}
	} elsif ($switch eq "exact") {
		$exact = 1;
	} elsif ($switch eq "export") {
		$exportField = shift;
		if (!grep(/^$exportField$/, @validFields)) {
			help("I can only export by $validFields.");
		}
	} elsif ($switch eq "folder") {
		if ($exportFolder = shift) {
			if (-e $exportFolder) {
				#if the folder exists, it needs to be a folder
				help("$exportFolder already exists and is not a folder.") if !-d $exportFolder;
			}
		} else {
			help("The folder option requires a folder name.");
		}
	} elsif ($switch eq "keep-time") {
		$keepTime = 1;
	} elsif ($switch eq "import") {
```

```
			if ($importDB = shift) {
				if ($importHandle = DBI->connect("dbi:SQLite:dbname=$importDB")) {
					createTable($importHandle, $dbTable, \%dbFields);
					$importHandle->{AutoCommit} = 0;
				} else {
					print "Unable to open database $importDB: $!\n";
				}
			} else {
				help("The import switch requires a database name to import to.");
			}
		} else {
			help("I do not understand the option '$switch'.");
		}
}
#the first item on the command line is what we're searching for
if (%searches) {
		#if we're looking for exact matches, set them up ahead of time
		if ($exact) {
			foreach $search (keys %searches) {
				$searchText = $searches{$search};
				$searches{$search} = "^$searchText\$";
			}
		}
		while (<>) {
			#split out the song information
			chomp;
			($song, $duration, $artist, $album, $year, $rating, $ripdate, $track, $genre) = split("\t");
			foreach $searchField (keys %searches) {
				$needle = $searches{$searchField};
				$haystack = $$searchField;
				$matched = match($haystack, $needle);
				last if !$matched;
			}
			#reverse the match if we want non-matching lines
			if ($reverse) {
				$matched = !$matched;
			}
			#print the information if this line is one we want
			if ($matched) {
				#maintain the timestamp if we need it
				if ($keepTime && $lastFile ne $ARGV) {
					@fileInfo = stat($ARGV);
					$fileMod = $fileInfo[9];
					$lastModified = $fileMod if $fileMod > $lastModified;
					$lastFile = $ARGV;
				}
				$matches++;
				if ($format eq "raw") {
					$text = "$_\n";
				} elsif ($format eq "html") {
					$text = "<tr><td>$song</td><td>$album</td><td>$artist</td></tr>\n";
				} elsif ($format eq "summary") {
					$artists{$artist}++;
				} else {
					$text = "$song ($album, by $artist)\n";
				}
				#store or print the display text and the sort text
				if ($sortby || $exportField) {
					$matches[$#matches+1]{'text'} = $text;
					$matches[$#matches]{'sort'} = $$sortby if $sortby;
					if ($exportField) {
						$filename = $$exportField;
						if ($filename ne "") {
							$filename = replace($filename, "/", "-");
						} else {
							$filename = "Unknown $exportField";
						}
						$matches[$#matches]{'file'} = $filename;
					}
				} else {
```

```
					print $text;
				}
				#import into a database?
				if ($importDB) {
					insertRow($importHandle, $dbTable, \%dbFields);
				}
			}
			last if $limit && $matches >= $limit;
		}
		#commit insertions if we are importing into a database
		$importHandle->commit;
		if (%artists) {
			@artists = keys %artists;
			@artists = sort byArtistCount @artists;
			foreach $artist (@artists) {
				$artistCount = $artists{$artist};
				print "$artist: $artistCount\n";
			}
		} elsif (@matches) {
			@matches = sort byCustom @matches if $sortby;
			#create a folder if necessary, and move into it
			if ($exportField && $exportFolder) {
				if (!-e $exportFolder) {
					if (!mkdir($exportFolder)) {
						print "Unable to create $exportFolder: $!\n";
						exit;
					}
				}
				if (!chdir($exportFolder)) {
					print "Unable to move into $exportFolder: $!\n";
					exit;
				}
			}
			foreach $match (@matches) {
				if ($exportField) {
					$filename = $$match{'file'};
					#open the file if we haven't already
					if (!$files{$filename}) {
						if (!open($files{$filename}, ">$filename")) {
							print "Unable to open $filename: $!\n";
							exit;
						}
					}
					$filehandle = $files{$filename};
					print $filehandle $$match{'text'};
				} else {
					print $$match{'text'};
				}
			}
			#close all open files
			foreach $filename (keys %files) {
				$filehandle = $files{$filename};
				close($filehandle);
				utime($lastModified, time(), $filename);
			}
		}
} else {
		help();
}
#describe how this script is used
sub help {
		#if there is an error message, print it out separated from the rest
		if (my($message) = shift) {
			print "\n$message\n\n";
		}
		print "Syntax: show [options] [song files]\n";
		print "\tSearch for some text in the song file. If no song file is specified\n";
		print "\t'show' will expect it on standard input.\n";
		print "\tA song file is a tab-delimited file with:\n";
		print "\ttitle, duration, artist, album, year, rating, rip date, track position, genre\n";
		print "\t--help: print this help text\n";
```

```
		print "\t--case: be sensitive to upper and lower case\n";
		print "\t--reverse: filter out songs that contain the search text\n";
		print "\t--limit x: limit to x results\n";
		print "\t--format <$validFormats>: choose format for results\n";
		print "\t--$validFields <searchtext>: search in the $validFields field\n";
		print "\t--sort <$validFields>: sort by specified field\n";
		print "\t--exact: the search text must match exactly\n";
		print "\t--export <$validFields>: export to files named after the specified field\n";
		print "\t--folder <foldername>: export files are created in the specified folder\n";
		print "\t--keep-time: keep the input file's timestamp on any exported files\n";
		print "\t--import <database name>: import data into a named database\n";
		print "At least one of the $validFields search requests must be specified.\n";
		exit;
}
sub byArtistCount {
		return $artists{$b} <=> $artists{$a};
}
sub englishJoin {
		my($punctuation) = shift;
		my($conjunction) = shift;
		my(@items) = @_;
		my($joined, $finalItem);
		if ($#items == -1) {
			$joined = "";
		} elsif ($#items == 0) {
			$joined = $items[0];
		} elsif ($#items == 1) {
			$joined = "$items[0] $conjunction $items[1]";
		} else {
			$finalItem = pop(@items);
			$joined = join($punctuation, @items) . "$punctuation$conjunction $finalItem";
		}
		return $joined;
}
sub match {
		my($searchIn) = shift;
		my($searchFor) = shift;
		my($matched) = 0;
		if ($sensitive) {
			$matched = $searchIn =~ /$searchFor/;
		} else {
			$matched = $searchIn =~ /$searchFor/i;
		}
		return $matched;
}
sub byCustom {
		if ($sensitive) {
			return $$a{'sort'} cmp $$b{'sort'};
		} else {
			return lc($$a{'sort'}) cmp lc($$b{'sort'});
		}
}
sub replace {
		my($text, $from, $to) = (shift, shift, shift);

		$text =~ s/$from/$to/g;
		return $text;
}
#expects a handle to a database connection, a name for a table, and an associative
#array of field names and field types passed as a reference
sub createTable {
		my($dbHandle) = shift;
		my($tableName) = shift;
		my($tableFields) = shift;
		my(@fieldCreators, $createQuery, $queryHandle, $fieldType);
		@tables = $importHandle->tables();
		if (!grep(/^"$dbTable"$/, @tables)) {
			#construct query to create table
			foreach $fieldName (keys %$tableFields) {
				$fieldType = $$tableFields{$fieldName};
				$fieldCreators[$#fieldCreators+1] = "$fieldName $fieldType";
```

```
				}
				$createQuery = "CREATE TABLE $tableName (" . join(", ", @fieldCreators) . ")";
				#tell database to create the table
				doQuery($dbHandle, $createQuery);
		}
}
#expects a handle to a database connection, a name for a table, and an associative
#array of field names and field types passed as a reference
#also expects that the field names match a currently-existing scalar variable
sub insertRow {
		my($dbHandle) = shift;
		my($tableName) = shift;
		my($tableFields) = shift;
		my(@fieldNames, @fieldValues, $fieldNames, $fieldValues, $fieldValue);
		my($insertQuery, $queryHandle);
		#create the field list and value list
		@fieldNames = keys %$tableFields;
		foreach $fieldName (@fieldNames) {
				$fieldValue = $$fieldName;
				#set $fieldValue to an appropriate SQL value
				if ($fieldValue eq "") {
						$fieldValue = NULL;
				} else {
						$fieldValue = '$dbHandle->quote($$fieldname);
				}
				$fieldValues[$#fieldValues+1] = $fieldValue;
		}
		$fieldNames = join(", ", @fieldNames);
		$fieldValues = join(", ", @fieldValues);
		$insertQuery = "INSERT INTO $tableName ($fieldNames) VALUES ($fieldValues)";
		#insert into the database
		doQuery($dbHandle, $insertQuery);
}
sub doQuery {
		my($dbHandle) = shift;
		my($query) = shift;
		if ($queryHandle = $dbHandle->prepare($query)) {
				if (!$queryHandle->execute) {
						print "Unable to execute $query: $!\n";
				}
		} else {
				print "Unable to prepare $query: $!\n";
				exit;
		}
}
```

Web CGIs

One of the most common uses of Perl on databases is displaying the data in the database on the web. Perl comes with a special module for use as a web page generator, called "CGI". CGI stands for Common Gateway Interface. It's a way for web servers to pass data to programs such as Perl scripts. The CGI module makes heavy use of objects and methods. You can find out more about Perl's CGI module by typing "perldoc CGI" from a Unix command line.

I'm going to assume that you've created the "love" database from the previous example, called LoveSongs, and that you've put it somewhere accessible. You usually do *not* want to put your databases inside your web site. For example, if your web site is in a folder called "public_html", your CGI will go in that folder but your database should be outside of it. Otherwise, people can download your database directly, without having to go through your CGI.

Perl web scripts usually end in ".cgi". Create a file called "music.cgi":

```
#!/usr/bin/perl
#display data from a show-created SQLite music database, on the web

use CGI::Pretty;
use DBI;

$html = new CGI;

#start the web page
print $html->header;
print $html->start_html("Love Songs");

#print the top of the html
print $html->h1("Love Songs of the Seventies & Eighties");
print $html->p("Welcome to my web page of", $html->em("songs that mention love"), "in their
title.");

#finish the page
print $html->end_html;
```

I'm actually using CGI::Pretty, rather than CGI, because it is easier to debug your Perl scripts when you can see the HTML it creates. The default for CGI is to put all of the text on the same line, for one really huge line of HTML. CGI::Pretty puts your HTML on separate lines. With or without CGI::Pretty, it works the same, so you can experiment with using each of them. Just change CGI::Pretty to CGI and then back again.

We create a "new cgi" and assign it to $html. This will be an object that knows how to create HTML code. It knows how to create a header, it knows how to create a paragraph, and it knows how HTML pages end. Each of the HTML parts can take lists of other parts. So you can see that for the "h1" part (level one header), we just send it some text. But for the following paragraph, one of the pieces needs to be emphasized.

You'll still need to know HTML a bit when using the CGI module; each of the parts is the same name as their HTML counterpart.

Once you have this script created and saved, don't forget to make it executable by you:

```
chmod u+x music.cgi
```

And you'll also need to make it executable by "other". Web servers will need permission to read this file in order to run it:

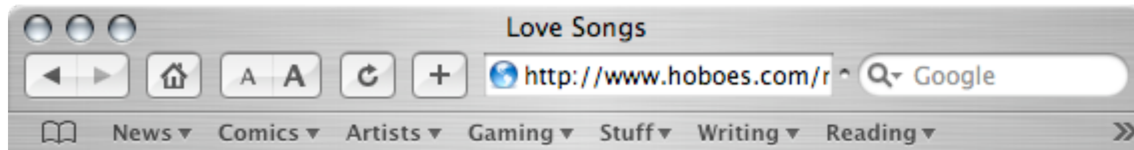
```
chmod o+x music.cgi
```

While you're at it, you'll need to make sure that the LoveSongs database is also accessible by "other":

```
chmod o+r /path/to/LoveSongs
```

You will need a web site that can run CGI scripts. This web site's server will also need the DBI module and the DBD::SQLite module installed. (If it's a server meant for web serving, it probably does.)

Upload it, and then view the CGI. You should get something like:



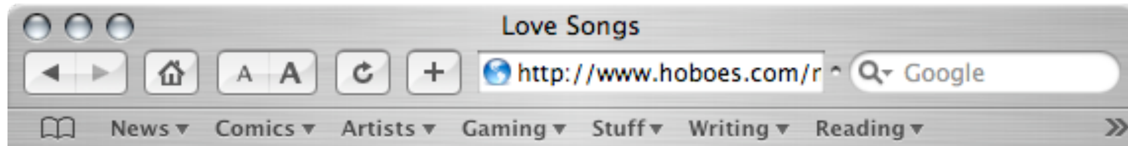
So, we can write web pages, we need to open the database and display it. Add some defaults to the top of the script, below the first comment:

```
@displayFields = ("song", "artist", "album", "year", "genre");
$dbFile = "/path/to/dbs/LoveSongs";
```

Below the paragraph and before the end of the page, add:

```
if ($dbHandle = DBI->connect("dbi:SQLite:dbname=$dbFile")) {
	print $html->table(
		$html->Tr($html->th(\@displayFields))
	);
} else {
	print $html->p($html->b("Problem opening database. Try again later: $!."));
}
```

We're opening the database just as we did before when we wanted to write to it. Then, we're printing out a table that contains one row; that row contains a series of header cells ("th"). Note that we're passing the list of fields as a reference. This is how $html->th() knows that each of these items needs its own cell. If they were passed as a normal list, $html->th() would put them all in the same cell, just as giving the paragraph multiple items put them all in the same paragraph.



Next step: display all of the rows. Replace the "$html->Tr(…)" line with:

```
$fields = join(", ", @displayFields);
$query = "SELECT $fields FROM music ORDER BY song";
if (@rows = getRows($dbHandle, $query)) {
	print $html->table(
		$html->Tr($html->th(\@displayFields), @rows)
	);
} else {
	print $html->p("Nothing found. Sorry.");
}
```

The query is going to look like "SELECT song, artist, album, year, genre FROM music ORDER BY song". We are going to pass that query to a subroutine called *getRows*.

If that subroutine returns something, we'll print the table just as we did before, but with the rows returned by *getRows*.

That subroutine will need to query the database and then go through each row that the database returns and turn them into HTML table rows.

```
sub getRows {
	my($db) = shift;
	my($query) = shift;

	my(@rows);
	if ($queryHandle = $db->prepare($query)) {
		if ($queryHandle->execute) {
			while ($row = $queryHandle->fetch) {
				$rows[$#rows+1] = $html->Tr($html->td($row));
```

```
                    }
                } else {
                    print $html->p("Unable to execute $query: $!");
                }
            } else {
                print $html->p("Unable to prepare $query: $!");
            }
            return @rows;
        }
```

The main difference between this and our other doQuery function is that, after executing the query this function also loops through each row, using:

```
        while ($row = $queryHandle->fetch) {
            $rows[$#rows+1] = $html->Tr($html->td($row));
        }
```

The *fetch* method on the query object gets the next row. More specifically, it gets a reference to a simple array of the items in the next row. Since the CGI module's HTML parts accept references as things to add the part to individually, calling $html->td($row) is like calling $html->td(\@row) if we had an @row list.

That's it. The web page should now display a list of about 292 songs.

Reference

Boolean logic

In case you've forgotten it from high school, here is the table of and, or, and exclusive or:

First Value	Second Value	AND	OR
True	True	True	True
True	False	False	True
False	False	False	False
False	True	False	True

Perl recognizes the following characters for use in *if*, *while*, or other such blocks for determining whether and how often the block executes:

<table>
<tr><th>Character</th><th>Meaning</th></tr>
<tr><td>&&</td><td>AND</td></tr>
<tr><td>||</td><td>OR</td></tr>
<tr><td>!</td><td>NOT</td></tr>
</table>

You can also use parentheses to force some logic to be interpreted before other logic. For example:

```
if ($isGod || ($isNietzsche && $godIsDead)) { ... }
```

will act on that *if* block if $isGod is true, or if both $isNietzsche is true and $godIsDead is true.

Comparison operators

Perl has two basic types of comparisons: comparisons between text and comparisons between numbers. Remember that Perl doesn't care whether a variable is text or a number until you use it in a manner that requires it to care.

Text Operator	Numeric Operator	Meaning
eq	==	are the two items equal?
ne	!=	are the two items unequal?
lt	<	does the left item come before the right item?
gt	>	does the left item come after the right item?
lte	<=	is the left item before or equal to the right item?
gte	>=	is the left item after or equal to the right item?
cmp	<=>	-1 if the left item comes first, +1 if the right item comes first, and 0 if both items are the same

File tests

There are a number of very useful tests you can perform on files. Remember that you'll usually want to determine if the file exists first, especially for the tests that return a value such as a file size or age.

Test	Tests
-e	the file exists
-r	the file is readable by your script
-w	the file is writable by your script
-x	the file is executable by your script
-s	the size of the file in bytes
-f	the file is a plain file, not a directory or alias
-d	the file is a directory
-M	the time since the file was last modified, in days

Tests can be combined by using the filename for the first test, and the underscore for subsequent tests. For example, to test if a file exists and has been modified less than 24 hours ago:

```
if (-e $filepath && -M _ < 1) { ... }
```

There are many more tests for more specialized purposes, but these should cover most of your needs.

Regular expressions

Remember that you can add an "i" to the end of a regular expression to make it ignore upper and lower case.

<table>
<tr><th>Character</th><th>Meaning</th></tr>
<tr><td>.</td><td>a single character</td></tr>
<tr><td>[characters]</td><td>a single character from a list or range of characters</td></tr>
<tr><td>[^characters]</td><td>a single character not from a list or range of characters</td></tr>
<tr><td>?</td><td>zero or one of the preceding piece of text</td></tr>
<tr><td>+</td><td>one or more of the preceding piece of text</td></tr>
<tr><td>*</td><td>zero or more of the preceding piece of text</td></tr>
<tr><td>(text)</td><td>group pieces of text for remembering later in $0 to $9 or for applying a ?, +, or *</td></tr>
<tr><td>|</td><td>choose between two or more options in a group of pieces</td></tr>
<tr><td>^</td><td>anchor the expression to the beginning of the line or text being searched</td></tr>
<tr><td>$</td><td>anchor the expression to the end of the line or text being searched</td></tr>
<tr><td>\character</td><td>backquote the next character so that it means what it is rather than its meaning as a regular expression.</td></tr>
</table>

For example, you might use /Dean ([A-Z]\.)?Martin/ to match Dean Martin, Dean M. Martin, Dean Q. Martin, or Dean Z. Martin.

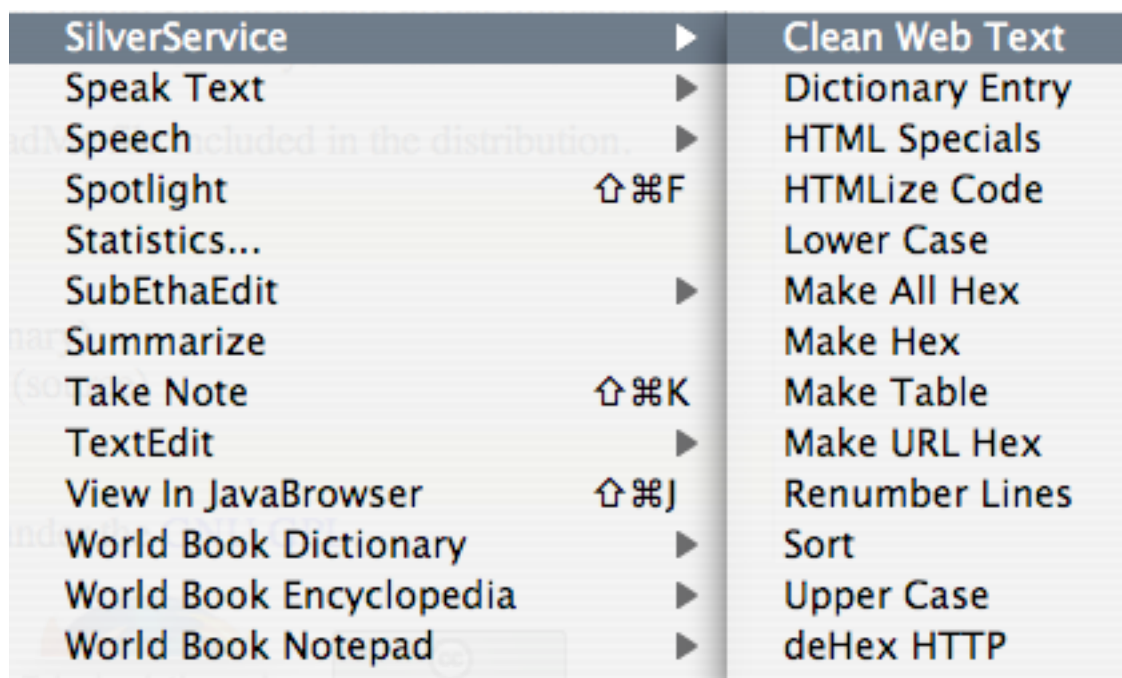
Use =~ to see if the item on the left matches the regular expression, and !~ to see if the item on the left does not match the regular expression.

SilverService

If you're using Mac OS X, I can't recommend *SilverService* strongly enough. You may be aware that applications can provide services to service-aware software. They'll show up in the "Services" submenu of every service-aware application's application menu and can act on that application's documents.

SilverService lets you put your scripts in that menu. Select text in a service-aware application (such as Smultron or Safari), choose your script from the Services menu, and the selected text will be piped to your script. Whatever your script prints out will replace your selected text. This is a great way to merge your command-line scripts with the Mac OS X GUI.

You can download SilverService from http://www.rho.org.uk/software/silverservice/



Note that SilverService is currently (a) abandoned, and (b) PowerPC, so it requires Rosetta. But it is open source, so someone may take it up again later. It works great up to at least Mac OS X 10.6 Snow Leopard.

More Information

The Perl web site at http://www.perl.org/ has documentation, tutorials, and links to several wonderful resources. O'Reilly has several very useful books about Perl. "Learning Perl" and "Programming Perl". Get "Learning Perl" if you need to learn programming, and get "Programming Perl" if you already know programming but want to learn programming in Perl.

If you're familiar with Perl but need some solutions, the "Perl Cookbook" is very useful. I have a review of it at http://www.hoboes.com/Mimsy/hacks/perl-cookbook/. If you look down towards the bottom of that page you'll also see one or two related Perl articles.

Finally, "The Perl Desktop Reference" is a great desktop—and pocket—reference to the Perl syntax and language.

> "The best book on programming for the layman is Alice in Wonderland; but that's because it's the best book on anything for the layman."

GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only

by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.
4. Preserve all the copyright notices of the Document.
5. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
6. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
7. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
8. Include an unaltered copy of this License.
9. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
10. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
11. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
12. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
13. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
14. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you

may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Perl

The Perl scripting language is ideal for creating cron scripts that deal with text, or that exchange data between two locations. It can easily filter text files, and also coordinate with other scripts and other commands on the Unix command line.

If you're using Mac OS X or Linux, you have Perl already installed. Learn the basics of this venerable and powerful scripting language. Automate your routine command-line tasks, and save time for writing more Perl scripts!