1

INTRODUCTION TO PERL ONE-LINERS

Perl one-liners are small and awesome Perl programs that fit in a single line of code. They do one thing really well—like changing line spacing, numbering lines, performing calculations, converting and substituting text, deleting and printing specific lines, parsing logs, editing files in-place, calculating statistics, carrying out system administration tasks, or updating a bunch of files at once. Perl one-liners will make you a shell warrior: what took you minutes (or even hours) to solve will now take you only seconds!

In this introductory chapter, I'll show you what one-liners look like and give you a taste of what's in the rest of the book. This book requires some Perl knowledge, but most of the one-liners can be tweaked and modified without knowing the language in depth.

Let's look at some examples. Here's one:

```
perl -pi -e 's/you/me/g' file
```

This one-liner replaces all occurrences of the text *you* with *me* in the file *file*. Very useful if you ask me. Imagine you're on a remote server and you need to replace text in a file. You can either open the file in a text editor and execute find-replace or simply perform the replacement through the command line and, bam, be done with it.

This one-liner and others in this book work well in UNIX. I'm using Perl 5.8 to run them, but they also work in newer Perl versions, such as Perl 5.10 and later. If you're on a Windows computer, you'll need to change them a little. To make this one-liner work on Windows, swap the single quotes for double quotes. To learn more about using Perl one-liners on Windows, see Appendix B.

I'll be using Perl's `-e` command-line argument throughout the book. It allows you to use the command line to specify the Perl code to be executed. In the previous one-liner, the code says "do the substitution (`s/you/me/g` command) and replace *you* with *me* globally (`/g` flag)." The `-p` argument ensures that the code is executed on every line of input and that the line is printed after execution. The `-i` argument ensures that *file* is edited in-place. Editing *in-place* means that Perl performs all the substitutions right in the file, overwriting the content you want to replace. I recommend that you always make a backup of the file you're working with by specifying the backup extension to the `-i` argument, like this:

```
perl -pi.bak -e 's/you/me/g' file
```

Now Perl creates a *file*`.bak` backup file first and only then changes the contents of *file*.

How about doing this same replacement in multiple files? Just specify the files on the command line:

```
perl -pi -e 's/you/me/g' file1 file2 file3
```

Here, Perl first replaces *you* with *me* in *file1* and then does the same in *file2* and *file3*.

You can also perform the same replacement only on lines that match `we`, as simply as this:

```
perl -pi -e 's/you/me/g if /we/' file
```

Here, you use the conditional `if /we/` to ensure that `s/you/me/g` is executed only on lines that match the regular expression `/we/`.

The regular expression can be anything. Say you want to execute the substitution only on lines with digits in them. You could use the `/\d/` regular expression to match numbers:

```
perl -pi -e 's/you/me/g if /\d/' file
```

How about finding all lines in a file that appear more than once?

```
perl -ne 'print if $a{$_}++' file
```

This one-liner records the lines you've seen so far in the `%a` hash and counts the number of times it sees the lines. If it has already seen the line, the condition `$a{$_}++` is true, so it prints the line. Otherwise it "automagically" creates an element that contains the current line in the `%a` hash and increments its value. The `$_` special variable contains the current line. This one-liner also uses the `-n` command-line argument to loop over the input, but unlike `-p`, it doesn't print the lines automatically. (Don't worry about all the command-line arguments right now; you'll learn about them as you work through this book!)

How about numbering lines? Super simple! Perl's `$.` special variable maintains the current line number. Just print it together with the line:

```
perl -ne 'print "$. $_"' file
```

You can do the same thing by using the `-p` argument and modifying the `$_` variable:

```
perl -pe '$_ = "$. $_"' file
```

Here, each line is replaced by the string `"$. $_"`, which is equal to the current line number followed by the line itself. (See one-liner 3.1 on page 17 for a full explanation.)

If you omit the filename at the end of the one-liner, Perl reads data from standard input. From now on, I'll assume the data comes from the standard input and drop the filename at the end. You can always put it back if you want to run one-liners on whole files.

You can also combine the previous two one-liners to create one that numbers only the repeated lines:

```
perl -ne 'print "$. $_" if $a{$_}++'
```

Another thing you can do is sum the numbers in each line using the `sum` function from the `List::Util` CPAN module. CPAN (Comprehensive Perl Archive Network; *http://www.cpan.org/*) is an archive of over 100,000

reusable Perl modules. `List::Util` is one of the modules on CPAN, and it contains various list utility functions. You don't need to install this module because it comes with Perl. (It's in Perl core.)

```
perl -MList::Util=sum -alne 'print sum @F'
```

The `-MList::Util` command-line argument imports the `List::Util` module. The `=sum` part of this one-liner imports the `sum` function from the `List::Util` module so that the program can use the function. Next, `-a` enables the automatic splitting of the current line into fields in the `@F` array. The splitting happens on the whitespace character by default. The `-l` argument ensures that `print` outputs a newline at the end of each line. Finally, `sum @F` sums all the elements in the `@F` list, and `print` prints the result followed by a newline (which I added with the `-l` argument). (See one-liner 4.2 on page 30 for a more detailed explanation.)

How about finding the date 1299 days ago? Try this:

```
perl -MPOSIX -le
  '@t = localtime; $t[3] -= 1299; print scalar localtime mktime @t'
```

I explain this example in detail in one-liner 4.19 (page 41), but basically you modify the fourth element of the structure returned by `localtime`, which happens to be days. You simply subtract 1299 days from the current day and then reassemble the result into a new time with `localtime mktime @t` and print the result in the scalar context to display human-readable time.

How about generating an eight-letter password? Here you go:

```
perl -le 'print map { ("a".."z")[rand 26] } 1..8'
```

The `"a".."z"` generates a list of letters from *a* to *z* (for a total of 26 letters). Then you randomly choose a letter eight times! (This example is explained in detail in one-liner 5.4 on page 51.)

Or suppose you want to find the decimal number that corresponds to an IP address. You can use `unpack` to find it really quickly:

```
perl -le 'print unpack("N", 127.0.0.1)'
```

This one-liner uses a *v-string*, which is a version literal. V-strings offer a way to compose a string with the specified ordinals. The IP address `127.0.0.1` is treated as a v-string, meaning the numbers `127`, `0`, `0`, `1` are concatenated together into a string of four characters, where the first character has ordinal value 127, the second and third characters have ordinal values 0, and the last character has ordinal value 1. Next, `unpack` unpacks them to a single decimal number in "network" (big-endian) order. (See one-liner 4.27 on page 45 for more.)

What about calculations? Let's find the sum of the numbers in the first column in a table:

```
perl -lane '$sum += $F[0]; END { print $sum }'
```

The lines are automatically split into fields with the `-a` argument, which can be accessed through the `@F` array. The first element of the array, `$F[0]`, is the first column, so you simply sum all the columns with `$sum += $F[0]`. When the Perl program finishes, it executes any code in the `END` block, which, in this case, prints the total sum. Easy!

Now let's find out how many packets have passed through `iptables` rules:

```
iptables -L -nvx | perl -lane '$pkts += $F[0]; END { print $pkts }'
```

The `iptables` program outputs the packets in the first column. All you have to do to find out how many packets have passed through the firewall rules is sum the numbers in the first column. Although `iptables` will output table headers as well, you can safely ignore these because Perl converts them to zero for the `+=` operation.

How about getting a list of all users on the system?

```
perl -a -F: -lne 'print $F[4]' /etc/passwd
```

Combining `-a` with the `-F` argument lets you specify the character where lines should be split, which, by default, is whitespace. Here, you split lines on the colon character, the record separator of `/etc/passwd`. Next, you print the fifth field, `$F[4]`, which contains the user's real name.

If you ever get lost with command-line arguments, remember that Perl comes with a fantastic documentation system called *perldoc*. Type **`perldoc perlrun`** at the command line. This will display the documentation about how to run Perl and all the command-line arguments. It's very useful when you suddenly forget which command-line argument does what and need to look it up quickly. You may also want to read *perldoc perlvar*, which explains variables; *perldoc perlop*, which explains operators; and *perldoc perlfunc*, which explains functions.

Perl one-liners let you accomplish many tasks quickly. You'll find over 130 one-liners in this book. Read them, try them, and soon enough you'll be the local shell wizard. (Just don't tell your friends—unless you want competition.)

Enjoy!