

Perl One-Liners Explained

by

[@pkrumins](#)

Peteris Krumins

peter@catonmat.net

<http://www.catonmat.net>

good coders code, great reuse

Contents

Contents	i
Preface	vi
1 Introduction	1
1.1 Perl One-Liners	1
2 Spacing	6
2.1 Double space a file	6
2.2 Double space a file, except the blank lines	8
2.3 Triple space a file	9
2.4 N-space a file	10
2.5 Add a blank line before every line	10
2.6 Remove all blank lines	11
2.7 Remove all consecutive blank lines, leaving just one	12
2.8 Compress/expand all blank lines into N consecutive ones	13
2.9 Double space all words	13
2.10 Remove all spacing between words	13
2.11 Change all spacing between words to one space	14
2.12 Insert a space between each character	14
3 Numbering	15
3.1 Number all lines in a file	15
3.2 Number only non-empty lines in a file	15
3.3 Number and print only non-empty lines in a file	16
3.4 Number all lines but print line numbers for only non-empty lines	16
3.5 Number only lines that match a pattern, print others un- modified	16

3.6	Number and print only lines that match a pattern	17
3.7	Number all lines, but print line numbers only for lines that match a pattern	17
3.8	Number all lines in a file using a custom format	17
3.9	Print the total number of lines in a file (emulate wc -l)	18
3.10	Print the number of non-empty lines in a file	19
3.11	Print the number of empty lines in a file	20
3.12	Print the number of lines in a file that match a pattern	21
3.13	Number words across all lines	21
3.14	Number words on each line individually	21
3.15	Replace all words with their numeric positions	22
4	Calculations	23
4.1	Check if a number is a prime	23
4.2	Print the sum of all the fields on a line	24
4.3	Print the sum of all the fields on all lines	25
4.4	Shuffle all fields of every line	25
4.5	Find the minimum element on every line	27
4.6	Find the minimum element over all the lines	27
4.7	Find the maximum element on a line	28
4.8	Find the maximum element over all the lines	28
4.9	Replace each field with its absolute value	29
4.10	Find the total number of fields (words) on each line	29
4.11	Print the total number of fields (words) on each line followed by the line	29
4.12	Find the total number of fields (words) on all lines	30
4.13	Print the total number of fields that match a pattern	30
4.14	Print the total number of lines that match a pattern	31
4.15	Print the number PI to n decimal places	31
4.16	Print the number E to n decimal places	31
4.17	Print UNIX time (seconds since Jan 1, 1970, 00:00:00 UTC)	32
4.18	Print GMT (Greenwich Mean Time) and local computer time	32
4.19	Print yesterday's date	34
4.20	Print date 14 months, 9 days and 7 seconds ago	34
4.21	Calculate factorial	35
4.22	Calculate greatest common divisor	35
4.23	Calculate least common multiple	36
4.24	Generate 10 random numbers between 5 and 15 (excluding 15)	37

4.25	Find and print all permutations of a list	37
4.26	Generate the power set	38
4.27	Convert an IP address to unsigned integer	39
4.28	Convert an unsigned integer to an IP address	40
5	String Creation and Array Creation	42
5.1	Generate and print the alphabet	42
5.2	Generate and print all the strings from "a" to "zz"	43
5.3	Create a hex lookup table	43
5.4	Generate a random 8 character password	44
5.5	Create a string of specific length	45
5.6	Create an array from a string	45
5.7	Create a string from an array	46
5.8	Find the numeric values for characters in a string	46
5.9	Convert a list of numeric ASCII values into a string	47
5.10	Generate an array with odd numbers from 1 to 100	47
5.11	Generate an array with even numbers from 1 to 100.	48
5.12	Find the length of the string	48
5.13	Find the number of elements in an array	48
6	Text Conversion and Substitution	49
6.1	ROT13 a string.	49
6.2	Base64 encode a string	50
6.3	Base64 decode a string	51
6.4	URL-escape a string	51
6.5	URL-unescape a string	51
6.6	HTML-encode a string	52
6.7	HTML-decode a string	52
6.8	Convert all text to uppercase	52
6.9	Convert all text to lowercase	53
6.10	Uppercase only the first word of each line	53
6.11	Invert the letter case	53
6.12	Camel case each line	54
6.13	Strip leading whitespace from the beginning of each line	55
6.14	Strip trailing whitespace from the end of each line	55
6.15	Strip whitespace from the beginning and end of each line	56
6.16	Convert UNIX newlines to DOS/Windows newlines	56
6.17	Convert DOS/Windows newlines to UNIX newlines	56

6.18	Convert UNIX newlines to Mac newlines	57
6.19	Substitute (find and replace) "foo" with "bar" on each line .	57
6.20	Substitute "foo" with "bar" on lines that match "baz" . . .	57
6.21	Reverse paragraphs in a file	58
6.22	Reverse all lines	58
7	Selective Printing and Deleting of Lines	60
7.1	Print the first line of a file (emulate head -1)	60
7.2	Print the first 10 lines of a file (emulate head -10)	60
7.3	Print the last line of a file (emulate tail -1)	61
7.4	Print the last 10 lines of a file (emulate tail -10)	62
7.5	Print only lines that match a regular expression	62
7.6	Print only lines that do not match a regular expression . . .	63
7.7	Print the line before a line that matches a regular expression	63
7.8	Print the line after a line that matches a regular expression .	64
7.9	Print lines that match regex AAA and regex BBB in any order	64
7.10	Print lines that don't match match regexes AAA and BBB .	64
7.11	Print lines that match regex AAA followed by BBB and CCC	65
7.12	Print lines that are 80 chars or longer	65
7.13	Print lines that are less than 80 chars in length	65
7.14	Print only line 13	66
7.15	Print all lines except line 27	66
7.16	Print only lines 13, 19 and 67	66
7.17	Print all lines between two regexes	67
7.18	Print all lines from line 17 to line 30	67
7.19	Print the longest line	67
7.20	Print the shortest line	68
7.21	Print all lines that contain a number	68
7.22	Find all lines that contain only a number	68
7.23	Print all lines that contain only characters	69
7.24	Print every second line	69
7.25	Print every second line, starting the second line.	69
7.26	Print all lines that repeat	70
7.27	Print all unique lines	70
8	Handy Regular Expressions	71
8.1	Match something that looks like an IP address	71
8.2	Test if a number is in range 0-255	71

8.3	Match an IP address	72
8.4	Check if the string looks like an email address	72
8.5	Check if the string is a decimal number	73
8.6	Check if a word appears twice in the string	75
8.7	Increase all numbers by one in the string	75
8.8	Extract HTTP User-Agent string from the HTTP headers	75
8.9	Match printable ASCII characters	76
8.10	Match text between two HTML tags	76
8.11	Replace all tags with 	77
8.12	Extract all matches from a regular expression	77
9	perl1line.txt	78
A	Perl's Special Variables	93
A.1	Variable \$_	93
A.2	Variable \$.	96
A.3	Variables \$1, \$2, \$3,	97
A.4	Variable \$/	98
A.5	Variable \$\	99
A.6	Variable \$,	99
A.7	Variable \$"	100
A.8	Variable @F	100
A.9	Variable @ARGV	101
A.10	Variable %ENV	102
	Index	103

Preface

Thanks!

Thank you for buying my "Perl One-Liners Explained" e-book! This is my third e-book and I based it on the "[Famous Perl One-Liners Explained](#)" article series that I wrote on my www.catonmat.net blog. I went through all the one-liners in the articles, improved explanations, fixed a lot of mistakes and typos, added a bunch of new one-liners, added an [Introduction to Perl one-liners](#) and a new chapter on [Perl special variables](#).

You might wonder why I called the article series that I based the book on "famous"? It's because I based this article series on two similar article series about awk and sed that I had written in the past. These article series are "[Famous Awk One-Liners Explained](#)" and "[Famous Sed One-Liners Explained](#)". Now why did I call these series "famous"? Well, because I based the articles on the popular [awk1line.txt](#) and [sed1line.txt](#) files by [Eric Pement](#). These files have been circulating around Unix newsgroups and forums for years and they're very popular among Unix programmers and sysadmins. That's how I actually really learned awk and sed myself. I went through all the one-liners in those files, tried them out and endeavored to understand exactly how they work. Then I decided that it would be a good idea to explain them on my blog, which I did, and after that I thought, why not create my own [perl1line.txt](#) file. So I did. I also included this file in the e-book as [Chapter 9](#).

As I mentioned, this is my third e-book. My first e-book is "[Awk One-Liners Explained](#)" and my second e-book is "[Sed One-Liners Explained](#)." All these e-books are written in the same style, so if you enjoy this e-book, you'll also enjoy the other two!

I have planned writing many more e-books. The next few are going to be the [catonmat](#) book, a practical guide to vim, and a practical guide to anonymity. If you're interested, [subscribe to my blog](#), [follow me on Twitter](#) or [follow me on Google+](#). That way you'll be the first to know when I publish it!

Enjoy!

One

Introduction

1.1 Perl One-Liners

Perl one-liners are small and awesome Perl programs that fit in a single line of code and they do one thing really well. These things include changing line spacing, numbering lines, doing calculations, converting and substituting text, deleting and printing certain lines, parsing logs, editing files in-place, doing statistics, carrying out system administration tasks, updating a bunch of files at once, and many more. Perl one-liners will make you the shell warrior. Anything that took you minutes to solve, will now take you seconds!

Let's look at several examples to get more familiar with one-liners. Here is 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 are on a remote server and have this file and you need to do the replacement. You can either open it in text editor and execute find-replace or just do it through command line and, bam, be done with it.

The **-e** argument is the best argument. It allows you to specify the Perl code to be executed right on the command line. In this one-liner the code says, do the substitution (**s/.../.../** command) and replace **you** with **me** globally (**/g** flag). The **-p** argument makes sure the code gets executed on every line, and that the line gets printed out after that. The **-i** argument makes sure that **file** gets edited in-place, meaning Perl opens the file, executes the substitution for each line, and puts it back in the file.

Don't worry too much about the command line arguments right now, you'll learn all about them as you work through the book!

How about doing the same replacement in multiple files? Just specify them on the command line!


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

Now let's do the same replacement only on lines that match `we`? It's as simple as this:

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

Here we use the conditional `if /we/`. This makes sure that `s/you/me/g` gets executed only on lines that match the regular expression `/we/`. The regular expression here can be anything. Let's say you want to execute the substitution only on lines that have digits on them. You can then use the `/\d/` regular expression that matches numbers:

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

Now how about finding all repeated lines in a file?

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

This one-liner records the lines seen so far in the `%a` hash and keeps the counter of how many times it has seen the lines. The `$a{$_}++` creates elements in the `%a` hash automatically. When it sees a repeated line, the value of that hash element is greater than one, and `if $a{$_}` is true, so it prints the line. This one-liner also uses the `-n` command line argument that loops over the input but unlike `-p` doesn't print the lines automatically. You'll find much more detailed description of `-n` in the first chapter of the book.

How about numbering lines? Super simple! Perl has the `$.` special variable that maintains the current line number. You can just print it out together with the line:

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

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

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

Here each line gets replaced by the string "\$. \$_", which is the current line number followed by the line itself. See [3.1](#) for a full explanation of this one-liner.

How about we combine the previous two one-liners and create one that numbers repeated lines? Here we go:

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

Now let's do something different. Let's sum up all the numbers in each line. We'll use the `sum` function from the `List::Util` CPAN module. You can install it as easily as running `perl -MCPAN -e'install List::Util'`.

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

The `-MList::Util` command line argument imports the `List::Util` module, and the `=sum` part of it imports the `sum` function from it. Next `-a` enables automatic splitting of fields into the `@F` array. The `-l` argument makes sure that `print` outputs a newline at the end of each line. Finally the `sum @F` sums up all the elements in the `@F` list and `print` prints it out, followed by a newline (that was added by the `-l` argument). This one-liner is explained in more details in [section 4.2](#).

How about finding the date 1299 days ago? That's also solvable by a simple one-liner:

```
perl -MPOSIX -le '  
    @now = localtime;  
    $now[3] -= 1299;  
    print scalar localtime mktime @now  
'
```

This one-liner didn't quite fit in one line, but that's just because this book has large margins. I explain this example in great detail in one-liner [4.19](#). Basically we modify the 4th element of the structure returned by `localtime`, which happens to be days. So we just subtract 1299 days from the current day. Then we reassembles it into a new time through `localtime mktime @now` and print it in scalar context that prints human readable time. This one-liner is explained in more details in [sections 4.18](#), [4.19](#) and [4.20](#).

How about generating an 8 letter password? Again, solvable elegantly with a one-liner:

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

The `a..z` generates a list of letters from a to z (total 26). Then we randomly choose one of them, and we repeat it 8 times! This example is explained in much more detail in one-liner [5.4](#).

Here is another one. Suppose you want to quickly find the decimal number that corresponds to an IP address. You can use the `unpack` function and find it really quickly:

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

This one-liner uses a `vstring`, which is a version literal. The IP address `127.0.0.1` is treated as a `vstring`, which is basically the numbers `127, 0, 0, 1` concatenated together. Next the `unpack` function unpacks them to a single decimal number. A much more detailed explanation is given in one-liner [4.27](#).

Now how about calculations? Let's find the sum of the numbers in the first column:

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

Here the lines automatically get split up into fields through the `-a` argument. The fields can be now accessed through the `@F` array. The first element of the array, `$F[0]`, is the first column. So all we have to do is sum all the columns up through `$sum += $F[0]`. When the Perl program ends its job, it executes any code in the special `END` block. In this case we print out the total sum there. Really easy!

Now let's find out how many packets have gone through all the `iptables` rules. It's this simple:

```
iptables -L -nvx | perl -lane '  
  $_ = $F[0]; $pkts += $_ if /\^d/; END { print $pkts }  
'
```

The iptables program outputs the packets as the first field. All we do is check if the first field is numeric (because it also outputs label header), and if so, sum the packets up, just like in the previous one-liner.

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

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

Combining `-a` with `-F` argument allows you to specify the character that lines should be split on. In this case it's the column, which is the record separator of `/etc/passwd`. Next we just print the 5th field `$F[4]`, which is the real name of the user. Really quick and easy.

As you can see, knowing Perl one-liners lets you accomplish many tasks quickly. Overall this e-book has 130 unique one-liners, which will let you become the shell wizard. Many one-liners are presented in several different ways so the total number of one-liners in this book is over 200. Enjoy!

Two

Spacing

2.1 Double space a file

```
perl -pe '$\="\n"'
```

This one-liner double spaces a file. There are three things to explain in this one-liner. The `-p` and `-e` command line options, and the `$_` variable.

First let's start with the `-e` option. The `-e` option can be used to enter a Perl program directly in the command line. Typically you don't want to create source files for every small program. With `-e` you can easily write the program right in the command line as a one-liner.

Next the `-p` switch. Specifying `-p` to a Perl program causes it to assume the following loop around your program:

```
while (<>) {
    # your program goes here (specified by -e)
} continue {
    print or die "-p failed: $_\n";
}
```

Broadly speaking, this construct loops over all the input, executes your code and prints the value of `$_`. This way you can quickly modify all or some lines of the input. The `$_` variable can be explained as an anonymous variable that gets filled with the current line of text. (You'll see later that it can be filled with other stuff as well.).

However, it's important to understand what is going on in this loop in more details. First the `while (<>)` loop takes each line from the standard input and puts it in the `$_` variable. Next the code specified by `-e` gets executed. Then the `print or die` part gets executed. The `continue` statement executes the `print or die` statement after each line. The `print or die` tries to print the contents of the `$_` variable. If it fails (for example, the

This is a preview copy (first 13 pages)

Get the full e-book at:

www.catonmat.net/blog/perl-book/