learnbyexample

Perl one-liners cookbook

```
perl -nE 'say $& if /\d+$/'

perl -ne 'print if !$h{$_}++'

perl -pe 's/\w+/reverse $&/ge'

perl -anE 'say $F[-2]'
```

Sundeep Agarwal

Table of contents

Preface

This book focuses on Perl usage from the command line, similar to `grep` , `sed` and `awk` usage. Syntax and features of these tools (along with languages like `C` and `bash`) were inspirations for Perl, so prior experience with them would make it easier to learn Perl.

You'll learn about various command line options and Perl features that make it possible to write compact cli scripts. Learning to use Perl from the command line will also allow you to construct solutions where Perl is just another tool in the shell ecosystem.

Prerequisites

You should be comfortable with programming basics and have prior experience working with Perl. You should know concepts like scalar, array, hash and special variables, be familiar with control structures, regular expressions etc. If you need resources to get started with Perl and regular expressions, you can start with these links:

- perldoc: perlintro
- learnxinyminutes: perl
- perldoc: perlretut

You should also have prior experience working with command line, `bash` shell and be familiar with concepts like file redirection, command pipeline and so on.

Conventions

- The examples presented here have been tested with **Perl version 5.32.0** and includes features not available in earlier versions.
- Code snippets shown are copy pasted from **bash** shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines have been added to improve readability, only `real` time is shown for speed comparisons and so on.
- Unless otherwise noted, all examples and explanations are meant for **ASCII** characters
  - See also stackoverflow: why does modern perl avoid utf-8 by default
- External links are provided for further reading throughout the book. Not necessary to immediately visit them. They have been chosen with care and would help, especially during re-reads.
- The learn_perl_oneliners repo has all the code snippets and files used in examples and exercises and other details related to the book. If you are not familiar with `git` command, click the **Code** button on the webpage to get the files.

Acknowledgements

- perl documentation — manuals, tutorials and examples
- /r/perl/ — helpful forum for beginners and experienced programmers alike
- stackoverflow — for getting answers to pertinent questions on Perl, one-liners, etc
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- LibreOffice Draw — cover image
- pngquant and svgcleaner for optimizing images
- Warning and Info icons by Amada44 under public domain
- softwareengineering.stackexchange and skolakoda for programming quotes

A heartfelt thanks to all my readers. Your valuable support has significantly eased my financial concerns and allows me to continue writing books.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/learn_perl_oneliners/issues

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample. He has also been a technical reviewer for Command Line Fundamentals book and video course published by Packt.

List of books: https://learnbyexample.github.io/books/

License

Book version

1.5

See Version_changes.md to track changes across book versions.

One-liner introduction

This chapter will give an overview of `perl` syntax for command line usage and some examples to show what kind of problems are typically suited for one-liners.

Why use Perl for one-liners?

I assume you are already familiar with use cases where command line is more productive compared to GUI. See also this series of articles titled Unix as IDE.

A shell utility like `bash` provides built-in commands and scripting features to easily solve and automate various tasks. External *nix commands like `grep` , `sed` , `awk` , `sort` , `find` , `parallel` , etc can be combined to work with each other. Depending upon your familiarity with those tools, you can either use `perl` as a single replacement or complement them for specific use cases.

Here's some one-liners (options will be explained later):

- `perl -pe 's/(?:\x27;\x27|";")(*SKIP)(*F)|;/#/g'` — change `;` to `#` but don't change `;` within single or double quotes
- `perl -MList::Util=uniq -e 'print uniq <>'` — retain only first copy of duplicated lines, uses built-in module `List::Util`
- `perl -MRegexp::Common=net -nE 'say $& while /$RE{net}{IPv4}/g'` — extract only IPv4 addresses, using a third-party Regexp::Common module
- Some stackoverflow Q&A that I've answered over the years with simpler `perl` solution compared to other cli tools
  - replace string with incrementing value
  - sort rows in csv file without header & first column
  - reverse matched pattern
  - append zeros to list
  - arithmetic replacement in a text file
  - reverse complement DNA sequence for a specific field

The selling point of `perl` over tools like `grep` , `sed` and `awk` includes feature rich regular expression engine and standard/third-party modules. If you don't already know the syntax and idioms for `sed` and `awk` , learning command line options for `perl` would be the easier option. Another advantage is that `perl` is more portable, given the many differences between GNU, BSD, Mac and other such implementations. The main disadvantage is that `perl` is likely to be verbose and slower for features that are supported out of the box by those tools.

See also unix.stackexchange: when to use grep, sed, awk, perl, etc

Installation and Documentation

If you are on a Unix like system, you are most likely to already have some version of Perl installed. See cpan: Perl Source for instructions to install the latest `perl` version from source. `perl v5.32.0` is used for all the examples shown in this book.

You can use `perldoc` command to access documentation from the command line. You can visit https://perldoc.perl.org/ if you wish to read it online, which also has a handy search feature. Here's some useful links to get started:

- perldoc: overview
- perldoc: perlintro
- perldoc: faqs

Command line options

`perl -h` gives the list of all command line options, along with a brief description. See perldoc: perlrun for documentation on these command switches.

Option	Description
`-0[octal]`	specify record separator (`\0` , if no argument)
`-a`	autosplit mode with `-n` or `-p` (splits `$_` into `@F`)
`-C[number/list]`	enables the listed Unicode features
`-c`	check syntax only (runs `BEGIN` and `CHECK` blocks)
`-d[:debugger]`	run program under debugger
`-D[number/list]`	set debugging flags (argument is a bit mask or alphabets)
`-e program`	one line of program (several `-e` 's allowed, omit programfile)
`-E program`	like `-e` , but enables all optional features
`-f`	don't do `$sitelib/sitecustomize.pl` at startup
`-F/pattern/`	`split()` pattern for `-a` switch (`//` 's are optional)
`-i[extension]`	edit `<>` files in place (makes backup if extension supplied)
`-Idirectory`	specify `@INC/#include` directory (several `-I` 's allowed)
`-l[octal]`	enable line ending processing, specifies line terminator
`-[mM][-]module`	execute `use/no module...` before executing program
`-n`	assume `while (<>) { ... }` loop around program
`-p`	assume loop like `-n` but `print` line also, like `sed`
`-s`	enable rudimentary parsing for switches after programfile
`-S`	look for programfile using `PATH` environment variable
`-t`	enable tainting warnings
`-T`	enable tainting checks
`-u`	dump core after parsing program
`-U`	allow unsafe operations
`-v`	print version, patchlevel and license
`-V[:variable]`	print configuration summary (or a single `Config.pm` variable)
`-w`	enable many useful warnings
`-W`	enable all warnings
`-x[directory]`	ignore text before `#!perl` line (optionally `cd` to directory)
`-X`	disable all warnings

This chapter will show examples with `-e` , `-l` , `-n` , `-p` and `-a` options. Some more options will be covered in later chapters, but not all of them are discussed in this book.

Executing Perl code

If you want to execute a `perl` program file, one way is to pass the filename as argument to the `perl` command.

```
$ echo 'print "Hello Perl\n"' > hello.pl
$ perl hello.pl
Hello Perl
```

For short programs, you can also directly pass the code as an argument to the `-e` or `-E` options. See perldoc: feature for details about the features enabled by the `-E` option.

```
$ perl -e 'print "Hello Perl\n"'
Hello Perl

$ # multiple statements can be issued separated by ;
$ # -l option will be covered in detail later, appends \n to 'print' here
$ perl -le '$x=25; $y=12; print $x**$y'
59604644775390625
$ # or, use -E and 'say' instead of -l and 'print'
$ perl -E '$x=25; $y=12; say $x**$y'
59604644775390625
```

Filtering

`perl` one-liners can be used for filtering lines matched by a regexp, similar to `grep` , `sed` and `awk` . And similar to many command line utilities, `perl` can accept input from both `stdin` and file arguments.

```
$ # sample stdin data
$ printf 'gate\napple\nwhat\nkite\n'
gate
apple
what
kite

$ # print all lines containing 'at'
$ # same as: grep 'at' and sed -n '/at/p' and awk '/at/'
$ printf 'gate\napple\nwhat\nkite\n' | perl -ne 'print if /at/'
gate
what

$ # print all lines NOT containing 'e'
$ # same as: grep -v 'e' and sed -n '/e/!p' and awk '!/e/'
$ printf 'gate\napple\nwhat\nkite\n' | perl -ne 'print if !/e/'
what
```

By default, `grep` , `sed` and `awk` will automatically loop over input content line by line (with `\n` as the line distinguishing character). The `-n` or `-p` option will enable this feature for `perl` . O module section shows the code Perl runs with these options.

As seen before, the `-e` option accepts code as command line argument. Many shortcuts are available to reduce the amount of typing needed. In the above examples, a regular expression (defined by the pattern between a pair of forward slashes) has been used to filter the input. When the input string isn't specified, the test is performed against special variable `$_` , which has the contents of the current input line here (the correct term would be input **record**, see Record separators chapter). `$_` is also the default argument for many functions like `print` and `say` . To summarize:

- `/REGEXP/FLAGS` is a shortcut for `$_ =~ m/REGEXP/FLAGS`
- `!/REGEXP/FLAGS` is a shortcut for `$_ !~ m/REGEXP/FLAGS`

See perldoc: match for help on `m` operator. See perldoc: special variables for documentation on `$_` , `$&` , etc.

Here's an example with file input instead of `stdin` .

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

$ perl -nE 'say $& if /(?<!-)\d+$/' table.txt
42
14

$ # if the condition isn't required, capture groups can be used
$ perl -nE 'say /(\d+)$/' table.txt
42
7
14
```

The learn_perl_oneliners repo has all the files used in examples (like `table.txt` in the above example).

Substitution

Use `s` operator for search and replace requirements. By default, this operates on `$_` when the input string isn't provided. For these examples, `-p` option is used instead of `-n` option, so that the value of `$_` is automatically printed after processing each input line. See perldoc: search and replace for documentation and examples.

```
$ # for each input line, change only first ':' to '-'
$ # same as: sed 's/:/-/' and awk '{sub(/:/, "-")} 1'
$ printf '1:2:3:4\na:b:c:d\n' | perl -pe 's/:/-/'
1-2:3:4
a-b:c:d

$ # for each input line, change all ':' to '-'
$ # same as: sed 's/:/-/g' and awk '{gsub(/:/, "-")} 1'
$ printf '1:2:3:4\na:b:c:d\n' | perl -pe 's/:/-/g'
1-2-3-4
a-b-c-d
```

The `s` operator modifies the input string it is acting upon if the pattern matches. In addition, it will return number of substitutions made if successful, otherwise returns a *false* value (empty string or `0`). You can use `r` flag to return string after substitution instead of in-place modification. As mentioned before, this book assumes you are already familiar with `perl` regular expressions. If not, see perldoc: perlretut to get started.

Field processing

Consider the sample input file shown below with fields separated by a single space character.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
```

Here's some examples that is based on specific field rather than the entire line. The `-a` option will cause the input line to be split based on whitespaces and the field contents can be accessed using `@F` special array variable. Leading and trailing whitespaces will be suppressed, so there's no possibility of empty fields. More details is discussed in Default field separation section.

```
$ # print the second field of each input line
$ # same as: awk '{print $2}' table.txt
$ perl -lane 'print $F[1]' table.txt
bread
cake
banana

$ # print lines only if the last field is a negative number
$ # same as: awk '$NF<0' table.txt
$ perl -lane 'print if $F[-1] < 0' table.txt
blue cake mug shirt -7

$ # change 'b' to 'B' only for the first field
$ # same as: awk '{gsub(/b/, "B", $1)} 1' table.txt
$ perl -lane '$F[0] =~ s/b/B/g; print "@F"' table.txt
Brown bread mat hair 42
Blue cake mug shirt -7
yellow banana window shoes 3.14
```

See Output field separator section for details on using array variable inside double quotes.

BEGIN and END

You can use a `BEGIN{}` block when you need to execute something before input is read and a `END{}` block to execute something after all of the input has been processed.

```
$ # same as: awk 'BEGIN{print "---"} 1; END{print "%%%"}'
$ seq 4 | perl -pE 'BEGIN{say "---"} END{say "%%%"}'
---
1
2
3
4
%%%
```

ENV hash

When it comes to automation and scripting, you'd often need to construct commands that can accept input from user, file, output of a shell command, etc. As mentioned before, this book assumes `bash` as the shell being used. To access environment variables of the shell, you can use the special hash variable `%ENV` with the name of the environment variable as a string key.

Quotes won't be used around `hash` keys in this book. See stackoverflow: are quotes around hash keys a good practice in Perl? on possible issues if you don't quote the `hash` keys.

```
$ # existing environment variable
$ # output shown here is for my machine, would differ for you
$ perl -E 'say $ENV{HOME}'
/home/learnbyexample
$ perl -E 'say $ENV{SHELL}'
/bin/bash

$ # defined along with perl command
$ # note that the variable definition is placed before the shell command
$ word='hello' perl -E 'say $ENV{word}'
hello
$ # the characters are preserved as is
$ ip='hi\nbye' perl -E 'say $ENV{ip}'
hi\nbye
```

Here's another example when a regexp is passed as an environment variable content.

```
$ cat word_anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart

```

```
$ # assume 'r' is a shell variable that has to be passed to the perl command
$ r='\Bpar\B'
$ rgx="$r" perl -ne 'print if /$ENV{rgx}/' word_anchors.txt
apparent effort
two spare computers
```

You can also make use of the `-s` option to assign a `perl` variable.

```
$ r='\Bpar\B'
$ perl -sne 'print if /$rgx/' -- -rgx="$r" word_anchors.txt
apparent effort
two spare computers
```

As an example, see my repo ch: command help for a practical shell script, where commands are constructed dynamically.

Executing external commands

You can execute external commands using the `system` function. See perldoc: system for documentation and details like how string/list argument is processed before it is executed.

```
$ perl -e 'system("echo Hello World")'
Hello World

$ perl -e 'system("wc -w <word_anchors.txt")'
12

$ perl -e 'system("seq -s, 10 > out.txt")'
$ cat out.txt
1,2,3,4,5,6,7,8,9,10
```

Return value of `system` or special variable `$?` can be used to act upon exit status of command issued. As per documentation:

The return value is the exit status of the program as returned by the `wait` call. To get the actual exit value, shift right by eight

```
$ perl -E '$es=system("ls word_anchors.txt"); say $es'
word_anchors.txt
0
$ perl -E 'system("ls word_anchors.txt"); say $?'
word_anchors.txt
0

$ perl -E 'system("ls xyz.txt"); say $?'
ls: cannot access 'xyz.txt': No such file or directory
512
```

To save the result of an external command, use backticks or `qx` operator. See perldoc: qx for documentation and details like separating out `STDOUT` and `STDERR` .

```
$ perl -e '$words = `wc -w <word_anchors.txt`; print $words'
12

$ perl -e '$nums = qx/seq 3/; print $nums'
1
2
3
```

See also stackoverflow: difference between backticks, system, and exec

Summary

This chapter introduced some of the common options for `perl` cli usage, along with typical cli text processing examples. While specific purpose cli tools like `grep` , `sed` and `awk` are usually faster, `perl` has a much more extensive standard library and ecosystem. And you do not have to learn a lot if you are already comfortable with `perl` but not familiar with those cli tools. The next section has a few exercises for you to practice the cli options and text processing use cases.

Exercises

Exercise related files are available from exercises folder of learn_perl_oneliners repo.

All the exercises are also collated together in one place at Exercises.md. To see the solutions, visit Exercise_solutions.md.

a) For the input file `ip.txt` , display all lines containing `is` .

```
$ cat ip.txt
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

##### add your solution here
This game is good
Today is sunny
```

b) For the input file `ip.txt` , display first field of lines *not* containing `y` . Consider space as the field separator for this file.

```
##### add your solution here
Hello
This
12345
```

c) For the input file `ip.txt`, display all lines containing no more than 2 fields.

```
##### add your solution here
Hello World
12345
```

d) For the input file `ip.txt`, display all lines containing `is` in the second field.

```
##### add your solution here
Today is sunny
```

e) For each line of the input file `ip.txt`, replace first occurrence of `o` with `0`.

```
##### add your solution here
Hell0 World
H0w are you
This game is g0od
T0day is sunny
12345
Y0u are funny
```

f) For the input file `table.txt`, calculate and display the product of numbers in the last field of each line. Consider space as the field separator for this file.

```
$ cat table.txt
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14

##### add your solution here
-923.16
```

g) Append `.` to all the input lines for the given `stdin` data.

```
$ printf 'last\nappend\nstop\n' | ##### add your solution here
last.
append.
stop.
```

h) Use contents of `s` variable to display all matching lines from the input file `ip.txt`. Assume that `s` doesn't have any regexp metacharacters. Construct the solution such that there's at least one word character immediately preceding the contents of `s` variable.

```
$ s='is'

##### add your solution here
This game is good
```

i) Use `system` to display contents of filename present in second field (space separated) of the given input line.

```
$ s='report.log ip.txt sorted.txt'
$ echo "$s" | ##### add your solution here
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

$ s='power.txt table.txt'
$ echo "$s" | ##### add your solution here
brown bread mat hair 42
blue cake mug shirt -7
yellow banana window shoes 3.14
```

Line processing

Now that you are familiar with basic `perl` cli usage, this chapter will dive deeper into line processing examples. You'll learn various ways for matching lines based on regular expressions, fixed string matching, line numbers, etc. You'll also see how to group multiple statements and learn about control flow keywords `next` and `exit` .

Regexp based filtering

As mentioned before:

- `/REGEXP/FLAGS` is a shortcut for `$_ =~ m/REGEXP/FLAGS`
- `!/REGEXP/FLAGS` is a shortcut for `$_ !~ m/REGEXP/FLAGS`

If required, you can also use different delimiters. Quoting from perldoc: match:

> If `/` is the delimiter then the initial `m` is optional. With the `m` you can use any pair of non-whitespace (ASCII) characters as delimiters. This is particularly useful for matching path names that contain `/` , to avoid LTS (**leaning toothpick syndrome**). If `?` is the delimiter, then a match-only-once rule applies, described in `m?PATTERN?` below. If `'` (single quote) is the delimiter, no variable interpolation is performed on the *PATTERN*. When using a delimiter character valid in an identifier, whitespace is required after the `m` . *PATTERN* may contain variables, which will be interpolated every time the pattern search is evaluated, except for when the delimiter is a single quote.

```
$ cat paths.txt
/foo/a/report.log
/foo/y/power.log
/foo/abc/errors.log

$ perl -ne 'print if /\/foo\/a\//' paths.txt
/foo/a/report.log

$ perl -ne 'print if m{/foo/a/}' paths.txt
/foo/a/report.log

$ perl -ne 'print if !m#/foo/a/#' paths.txt
/foo/y/power.log
/foo/abc/errors.log
```

Extracting matched portions

You can use regexp related special variables to extract only the matching portions instead of filtering entire matching line. Consider this input file.

```
$ cat programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

```

```
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

Here's some examples of extracting only the matched portion(s).

```
$ # note that this will print only the first match for each input line
$ perl -nE 'say $& if /\bt\w*[et]\b/' programming_quotes.txt
twice
the
that

$ perl -nE 'say join "::", @{^CAPTURE} if /not (.+)y(.+)/i' programming_quotes.txt
smart enough to debug it b:: Brian W. Kernighan
affect the way ::ou think about programming,
worth knowing b:: Alan Perlis

$ # sometimes capture groups are enough, you don't need special variables
$ # @{^CAPTURE} isn't needed here, as it is assumed that every line has a match
$ perl -nE 'say /^(\w+ ).*?(\d+)$/' table.txt
brown 42
blue 7
yellow 14
$ # or add a custom separator
$ perl -nE 'say join ":", /^(\w+).*?(\d+)$/' table.txt
brown:42
blue:7
yellow:14
```

Transliteration

The transliteration operator `tr` (or `y`) allows you to specify per character transformation rule. See perldoc: tr for documentation.

```
$ # rot13
$ echo 'Uryyb Jbeyq' | perl -pe 'tr/a-zA-Z/n-za-mN-ZA-M/'
Hello World

$ # use 'c' option to complement specified characters
$ echo 'foo:123:baz' | perl -pe 'tr/0-9\n/-/c'
----123----

$ # use 'd' option to delete specified characters
$ echo 'foo:123:baz' | perl -pe 'tr/0-9\n//cd'
123
```

```
$ # use 's' option to squeeze repeated characters
$ echo 'APPLE gobbledygook' | perl -pe 'tr|A-Za-z||s'
APLE gobledygok
$ # transliterate as well as squeeze
$ echo 'APPLE gobbledygook' | perl -pe 'tr|A-Z|a-z|s'
aple gobbledygook
```

Similar to `s` operator, `tr` will return number of changes made. Use `r` option to prevent in-place modification and return the transliterated string instead.

```
$ # match lines containing 'b' 2 times
$ perl -ne 'print if tr/b// == 2' table.txt
brown bread mat hair 42

$ s='orange apple appleseed'
$ echo "$s" | perl -pe 's#\bapple\b(*SKIP)(*F)|\w+#$&=~tr/a-z/A-Z/r#ge'
ORANGE apple APPLESEED
```

See also:

- stackoverflow: reverse complement DNA sequence for a specific field
- unix.stackexchange: count the number of characters except specific characters
- unix.stackexchange: scoring DNA data

Conditional substitution

These examples combine line filtering and substitution in different ways. As noted before, `s` operator will modify the input string and the return value can be used to know how many substitutions were made. Use the `r` flag to prevent in-place modification and get string output after substitution, if any.

```
$ # change commas to hyphens if the input line does NOT contain '2'
$ # prints all input lines even if substitution fails
$ printf '1,2,3,4\na,b,c,d\n' | perl -pe 's/,/-/g if !/2/'
1,2,3,4
a-b-c-d

$ # prints filtered input lines, even if substitution fails
$ perl -ne 'print s/by/**/rg if /not/' programming_quotes.txt
** definition, not smart enough to debug it ** Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing ** Alan Perlis

$ # print only if substitution succeeded
$ perl -ne 'print if s/1/one/g' programming_quotes.txt
naming things, and off-by-one errors by Leon Bambrick
```

Multiple conditions

It is good to remember that Perl is a programming language. You have control structures and you can combine multiple conditions using logical operators. You don't have to create a single complex regexp.

```
$ perl -ne 'print if /not/ && !/it/' programming_quotes.txt
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

$ perl -ane 'print if /twice/ || $#F > 11' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Some people, when confronted with a problem, think - I know, I will

$ perl -ne 'print if /s/ xor /m/' table.txt
brown bread mat hair 42
yellow banana window shoes 3.14
```

next

When `next` is executed, rest of the code will be skipped and the next input line will be fetched for processing. It doesn't affect `BEGIN` or `END` blocks as they are outside the file content loop.

```
$ perl -nE 'if(/\bpar/){print "%% $_"; next}
            say /s/ ? "X" : "Y"' word_anchors.txt
%% sub par
X
Y
X
%% cart part tart mart
```

Note that `{}` is used in the above example to group multiple statements to be executed for a single `if` condition. You'll see many more examples with `next` in coming chapters.

exit

The `exit` function is useful to avoid processing unnecessary input content when a termination condition is reached. See perldoc: exit for documentation.

```
$ # quits after an input line containing 'you' is found
$ perl -ne 'print; exit if /you/' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
$ # matching line won't be printed in this case
$ perl -pe 'exit if /you/' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
```

Use `tac` to get all lines starting from last occurrence of the search string with respect to entire file content.

```
$ tac programming_quotes.txt | perl -ne 'print; exit if /not/' | tac
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

You can optionally provide a status code as an argument to the `exit` function.

```
$ printf 'sea\neat\ndrop\n' | perl -ne 'print; exit(2) if /at/'
sea
eat
$ echo $?
2
```

Any code in `END` block will still be executed before exiting. This doesn't apply if `exit` was called from the `BEGIN` block.

```
$ perl -pE 'exit if /cake/' table.txt
brown bread mat hair 42

$ perl -pE 'exit if /cake/; END{say "bye"}' table.txt
brown bread mat hair 42
bye

$ perl -pE 'BEGIN{say "hi"; exit; say "hello"} END{say "bye"}' table.txt
hi
```

⚠ Be careful if you want to use `exit` with multiple input files, as `perl` will stop even if there are other files remaining to be processed.

Line number based processing

Line numbers can also be specified as a matching criteria using the `$.` special variable.

```
$ # print only the 3rd line
$ perl -ne 'print if $. == 3' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan

$ # print 2nd and 5th line
$ perl -ne 'print if $. == 2 || $. == 5' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
Some people, when confronted with a problem, think - I know, I will

$ # transliterate only 2nd line
$ printf 'gates\nnot\nused\n' | perl -pe 'tr/a-z/*/ if $. == 2'
gates
***
used

$ # print from particular line number to the end of input
$ seq 14 25 | perl -ne 'print if $. >= 10'
23
24
25
```

Use `eof` function to check for end of file condition. See perldoc: eof for documentation.

```
$ # same as: tail -n1 programming_quotes.txt
$ perl -ne 'print if eof' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick

$ perl -ne 'print "$.:$_" if eof' programming_quotes.txt
12:naming things, and off-by-1 errors by Leon Bambrick

$ # multiple file example
$ # same as: tail -q -n1 programming_quotes.txt table.txt
$ perl -ne 'print if eof' programming_quotes.txt table.txt
naming things, and off-by-1 errors by Leon Bambrick
yellow banana window shoes 3.14
```

For large input files, use `exit` to avoid processing unnecessary input lines.

```
$ seq 3542 4623452 | perl -ne 'if($. == 2452){print; exit}'
5993
$ seq 3542 4623452 | perl -ne 'print if $. == 250; if($. == 2452){print; exit}'
3791
5993

$ # here is a sample time comparison
$ time seq 3542 4623452 | perl -ne 'if($. == 2452){print; exit}' > f1
real    0m0.004s
$ time seq 3542 4623452 | perl -ne 'print if $. == 2452' > f2
real    0m0.740s
```

Range operator

You can use range operator to select between pair of matching conditions like line numbers and regexp. See perldoc: range for documentation.

```
$ # the range is automatically compared against $. in this context
$ # same as: perl -ne 'print if 3 <= $. <= 5'
$ seq 14 25 | perl -ne 'print if 3..5'
16
17
18

$ # the range is automatically compared against $_ in this context
$ # note that all the matching ranges are printed
$ perl -ne 'print if /are/ .. /by/' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan
There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

See Records bounded by distinct markers section for an alternate, flexible solution.

You can also mix line number and regexp conditions.

```
$ perl -ne 'print if 5 .. /use/' programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

$ # same logic as: perl -pe 'exit if /ll/'
$ # inefficient, but this will work for multiple file inputs
$ perl -ne 'print if !(/ll/ .. eof)' programming_quotes.txt table.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

brown bread mat hair 42
blue cake mug shirt -7
```

⚠ Both conditions can match the same line too! Also, if the second condition doesn't match, lines starting from first condition to the last line of the input will be matched.

```
$ # 'worth' matches the 9th line
$ perl -ne 'print if 9 .. /worth/' programming_quotes.txt
is not worth knowing by Alan Perlis

$ # there's a line containing 'affect' but doesn't have matching pair
$ # so, all lines till the end of input is printed
$ perl -ne 'print if /affect/ .. /XYZ/' programming_quotes.txt
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

Working with fixed strings

You can surround a regexp pattern with `\Q` and `\E` to match it as a fixed string, similar to `grep -F` option. `\E` can be left out if there's no further pattern to be specified. Variables are still interpolated, so if your fixed string contains `$` or `@` forming possible variables, you'll run into issues. For such cases, you can pass the string as an environment value and then apply `\Q` to that variable. See perldoc: quotemeta for documentation.

```
$ # no match, since [] are character class metacharacters
$ echo 'int a[5]' | perl -ne 'print if /a[5]/'

$ perl -E 'say "\Qa[5]"'
a\[5\]
$ echo 'int a[5]' | perl -ne 'print if /\Qa[5]/'
int a[5]
$ echo 'int a[5]' | perl -pe 's/\Qa[5]/b[12]/'
int b[12]
```

```
$ # $y and $z will be treated as variables here (default value empty string)
$ echo '$x = $y + $z' | perl -pe 's/\Q$y + $z/100/'
$x = $y100$z
$ echo '$x = $y + $z' | fs='$y + $z' perl -pe 's/\Q$ENV{fs}/100/'
$x = 100
$ # ENV is preferred since \\ is special in single quoted strings
$ perl -E '$x = q(x\y\\0z); say $x'
x\y\0z
$ x='x\y\\0z' perl -E 'say $ENV{x}'
x\y\\0z
```

If you just want to filter a line based on fixed string, you can also use the `index` function. This returns the matching position (which starts with `0`) and `-1` if the given string wasn't found. See perldoc: index for documentation.

```
$ echo 'int a[5]' | perl -ne 'print if index($_, "a[5]") != -1'
int a[5]
```

The above `index` example uses double quotes for the string argument, which allows escape sequences like `\t`, `\n`, etc and interpolation. This isn't the case with single quoted string values. Using single quotes within the script from command line requires messing with shell metacharacters. So, use `q` operator instead or pass the fixed string to be matched as an environment variable.

```
$ # double quotes allow escape sequences and interpolation
$ perl -E '$x=5; say "value of x:\t$x"'
value of x:     5

$ # use 'q' operator as an alternate to specify single quoted string
$ s='$a = 2 * ($b + $c)'
$ echo "$s" | perl -ne 'print if index($_, q/($b + $c)/) != -1'
$a = 2 * ($b + $c)

$ # or pass the string as environment variable
$ echo "$s" | fs='($b + $c)' perl -ne 'print if index($_, $ENV{fs}) != -1'
$a = 2 * ($b + $c)
```

You can use the return value of `index` function to restrict the matching to the start or end of the input line. The line content in `$_` variable contains the `\n` line ending character as well. You can either use `chomp` function explicitly or use the `-l` command line option, which will be discussed in detail in Record separators chapter. For now, it is enough to know that `-l` will remove the line ending from `$_` and add it back when `print` is used.

```
$ cat eqns.txt
a=b,a-b=c,c*d
a+b,pi=3.14,5e12
i*(t+9-g)/8,4-a+b

$ # start of line
$ s='a+b' perl -ne 'print if index($_, $ENV{s})==0' eqns.txt
a+b,pi=3.14,5e12
```

```
$ # end of line
$ # same as: s='a+b' perl -ne 'print if /\Q$ENV{s}\E$/' eqns.txt
$ # length function returns number of characters, by default acts on $_
$ # -l option is needed here to remove \n from $_
$ s='a+b' perl -lne '$pos = length() - length($ENV{s});
                     print if index($_, $ENV{s}) == $pos' eqns.txt
i*(t+9-g)/8,4-a+b
```

Here's some more examples using the return value of `index` function.

```
$ # since 'index' returns '-1' if there's no match,
$ # you need to add >=0 check as well for < or <= comparison
$ perl -ne '$i = index($_, "="); print if 0 <= $i <= 5' eqns.txt
a=b,a-b=c,c*d

$ # > or >= comparison is easy to specify
$ # if you use 3rd argument to 'index', you'll still have to check != -1
$ s='a+b' perl -ne 'print if index($_, $ENV{s})>=1' eqns.txt
i*(t+9-g)/8,4-a+b
```

If you need to match entire input line or field, you can use string comparison operators.

```
$ printf 'a.b\na+b\n' | perl -lne 'print if /^a.b$/'
a.b
a+b
$ printf 'a.b\na+b\n' | perl -lne 'print if $_ eq q/a.b/'
a.b
$ printf '1 a.b\n2 a+b\n' | perl -lane 'print if $F[1] ne q/a.b/'
2 a+b
```

To provide a fixed string in replacement section, environment variable comes in handy again. Or use `q` operator for directly providing the value, but you may have to workaround the delimiters being used and presence of `\\` characters.

```
$ # characters like $ and @ are special in replacement section
$ echo 'x+y' | perl -pe 's/\Qx+y/$x+@y/'
+

$ # provide replacement string as environment variable
$ echo 'x+y' | r='$x+@y' perl -pe 's/\Qx+y/$ENV{r}/'
$x+@y

$ # or, use 'e' flag to provide single quoted value as Perl code
$ echo 'x+y' | perl -pe 's/\Qx+y/q($x+@y)/e'
$x+@y

$ # need to workaround delimiters and \\ with 'q' operator based solution
$ echo 'x+y' | perl -pe 's/\Qx+y/q($x\/@y)/e'
$x/@y
$ echo 'x+y' | perl -pe 's|\Qx+y|q($x/@y)|e'
$x/@y
```

```
$ echo 'x+y' | perl -pe 's|\Qx+y|q($x/@y\\\z)|e'
$x/@y\\z
```

Summary

This chapter showed various examples of processing only lines of interest instead of entire input file. Filtering can be specified using a regexp, fixed string, line number or a combination of them. `next` and `exit` are useful to change the flow of code.

Exercises

a) Remove only the third line of given input.

```
$ seq 34 37 | ##### add your solution here
34
35
37
```

b) Display only fourth, fifth, sixth and seventh lines for the given input.

```
$ seq 65 78 | ##### add your solution here
68
69
70
71
```

c) For the input file `ip.txt`, replace all occurrences of `are` with `are not` and `is` with `is not` only from line number **4** till end of file. Also, only the lines that were changed should be displayed in the output.

```
$ cat ip.txt
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

##### add your solution here
Today is not sunny
You are not funny
```

d) For the given `stdin`, display only the first three lines. Avoid processing lines that are not relevant.

```
$ seq 14 25 | ##### add your solution here
14
15
16
```

e) For the input file `ip.txt`, display all lines from start of the file till the first occurrence of `game`.

```
##### add your solution here
Hello World
How are you
This game is good
```

f) For the input file `ip.txt`, display all lines that contain `is` but not `good`.

```
##### add your solution here
Today is sunny
```

g) For the input file `ip.txt`, extract the word before the whole word `is` as well as the word after it. If such a match is found, display the two words around `is` in reversed order. For example, `hi;1 is--234 bye` should be converted to `234:1`. Assume that whole word `is` will not be present more than once in a single line.

```
##### add your solution here
good:game
sunny:Today
```

h) For the given input string, replace `0xA0` with `0x7F` and `0xC0` with `0x1F`.

```
$ s='start address: 0xA0, func1 address: 0xC0'

$ echo "$s" | ##### add your solution here
start address: 0x7F, func1 address: 0x1F
```

i) Find the starting index of first occurrence of `is` or `the` or `was` or `to` for each input line of the file `idx.txt`. Assume all input lines will match at least one of these terms.

```
$ cat idx.txt
match after the last newline character
and then you want to test
this is good bye then
you were there to see?

##### add your solution here
12
4
2
9
```

j) Display all lines containing `[4]*` for the given `stdin` data.

```
$ printf '2.3/[4]*6\n2[4]5\n5.3-[4]*9\n' | ##### add your solution here
2.3/[4]*6
5.3-[4]*9
```

k) For the given input string, replace all lowercase alphabets to `x` only for words starting with `m`.

```
$ s='ma2T3a a2p kite e2e3m meet'
$ echo "$s" | ##### add your solution here
xx2T3x a2p kite e2e3m xxxx
```

l) For the input file `ip.txt`, delete all characters other than lowercase vowels and newline character. Perform this transformation only between a line containing `you` up to line number `4` (inclusive).

```
##### add your solution here
Hello World
oaeou
iaeioo
oaiu
12345
You are funny
```

In-place file editing

In the examples presented so far, the output from `perl` was displayed on the terminal or redirected to another file. This chapter will discuss how to write back the changes to the input file(s) itself using the `-i` command line option. This option can be configured to make changes to the input file(s) with or without creating a backup of original contents. When backups are needed, the original filename can get a prefix or a suffix or both. And the backups can be placed in the same directory or some other directory as needed.

With backup

You can use the `-i` option to write back the changes to the input file instead of displaying the output on terminal. When an extension is provided as an argument to `-i`, the original contents of the input file gets preserved as per the extension given. For example, if the input file is `ip.txt` and `-i.orig` is used, `ip.txt.orig` will be the backup filename.

```
$ cat colors.txt
deep blue
light orange
blue delight

$ # no output on terminal as -i option is used
$ # space is NOT allowed between -i and the extension
$ perl -i.bkp -pe 's/blue/-green-/' colors.txt
$ # changes are written back to 'colors.txt'
$ cat colors.txt
deep -green-
light orange
-green- delight

$ # original file is preserved in 'colors.txt.bkp'
$ cat colors.txt.bkp
deep blue
light orange
blue delight
```

Multiple input files are treated individually and the changes are written back to respective files.

```
$ cat t1.txt
have a nice day
bad morning
what a pleasant evening
$ cat t2.txt
worse than ever
too bad

$ perl -i.bkp -pe 's/bad/good/' t1.txt t2.txt
$ ls t?.*
t1.txt  t1.txt.bkp  t2.txt  t2.txt.bkp
```

```
$ cat t1.txt
have a nice day
good morning
what a pleasant evening
$ cat t2.txt
worse than ever
too good
```

Without backup

Sometimes backups are not desirable. Using `-i` option on its own will not create backups. Be careful though, as changes made cannot be undone. In such cases, test the command with sample input before using `-i` option on actual file. You could also use the option with backup, compare the differences with a `diff` program and then delete the backup.

```
$ cat fruits.txt
banana
papaya
mango

$ perl -i -pe 's/an/AN/g' fruits.txt
$ cat fruits.txt
bANANa
papaya
mANgo
```

Prefix backup name

A `*` character in the argument to `-i` option is special. It will get replaced with the input filename. This is helpful if you need to use a prefix instead of suffix for the backup filename. Or any other combination that may be needed.

```
$ ls *colors.txt*
colors.txt  colors.txt.bkp

$ # single quotes is used here as * is a special shell character
$ perl -i'bkp.*' -pe 's/-green-/yellow/' colors.txt
$ ls *colors.txt*
bkp.colors.txt  colors.txt  colors.txt.bkp
```

Place backups in different directory

The `*` trick can also be used to place the backups in another directory instead of the parent directory of input files. The backup directory should already exist for this to work.

```
$ mkdir backups
$ perl -i'backups/*' -pe 's/good/nice/' t1.txt t2.txt
$ ls backups/
t1.txt  t2.txt
```

Gory details of in-place editing

For more details about the `-i` option, see:

- effectiveperlprogramming: In-place editing gets safer in v5.28
- perldoc: -i option — documentation and underlying code
- perldoc faq: Why does Perl let me delete read-only files? Why does -i clobber protected files? Isn't this a bug in Perl?

Summary

This chapter discussed about the `-i` option which is useful when you need to edit a file in-place. This is particularly useful in automation scripts. But, do ensure that you have tested the `perl` command before applying to actual files if you need to use this option without creating backups.

Exercises

a) For the input file `text.txt` , replace all occurrences of `in` with `an` and write back the changes to `text.txt` itself. The original contents should get saved to `text.txt.orig`

```
$ cat text.txt
can ran want plant
tin fin fit mine line

##### add your solution here

$ cat text.txt
can ran want plant
tan fan fit mane lane
$ cat text.txt.orig
can ran want plant
tin fin fit mine line
```

b) For the input file `text.txt` , replace all occurrences of `an` with `in` and write back the changes to `text.txt` itself. Do not create backups for this exercise. Note that you should have solved the previous exercise before starting this one.

```
$ cat text.txt
can ran want plant
tan fan fit mane lane

##### add your solution here

$ cat text.txt
cin rin wint plint
tin fin fit mine line
$ diff text.txt text.txt.orig
1c1
< cin rin wint plint
---
> can ran want plant
```

c) For the input file `copyright.txt` , replace `copyright: 2018` with `copyright: 2020` and write back the changes to `copyright.txt` itself. The original contents should get saved to `2018_copyright.txt.bkp`

```
$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018

##### add your solution here

$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2020
$ cat 2018_copyright.txt.bkp
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
```

d) In the code sample shown below, two files are created by redirecting output of `echo` command. Then a `perl` command is used to edit `b1.txt` in-place as well as create a backup named `bkp.b1.txt` . Will the `perl` command work as expected? If not, why?

```
$ echo '2 apples' > b1.txt
$ echo '5 bananas' > -ibkp.txt
$ perl -ibkp.* -pe 's/2/two/' b1.txt
```

Field separators

This chapter will dive deep into field processing. You'll learn how to set input and output field separators, how to use regexps for defining fields and how to work with fixed length fields.

Default field separation

By default, the `-a` option splits based on one or more sequence of **whitespace** characters. In addition, whitespaces at the start or end of input gets trimmed and won't be part of field contents. Using `-a` is equivalent to `@F = split` . From perldoc: split:

> `split` emulates the default behavior of the command line tool `awk` when the PATTERN is either omitted or a string composed of a single space character (such as `' '` or `"\x20"` , but not e.g. `/ /`). In this case, any leading whitespace in EXPR is removed before splitting occurs, and the PATTERN is instead treated as if it were `/\s+/` ; in particular, this means that any contiguous whitespace (not just a single space character) is used as a separator. However, this special treatment can be avoided by specifying the pattern `/ /` instead of the string `" "` , thereby allowing only a single space character to be a separator.

```
$ # $#F gives index of last element, i.e. size of array - 1
$ echo '   a   b   c   ' | perl -anE 'say $#F'
2
$ # note that leading whitespaces isn't part of field content
$ echo '   a   b   c   ' | perl -anE 'say $F[0]'
a
$ # note that trailing whitespaces isn't part of field content
$ echo '   a   b   c   ' | perl -anE 'say "$F[-1]."'
c.

$ # here's another example with more whitespace characters thrown in
$ # in scalar context, @F will return size of the array
$ printf '     one \t\f\v two\t\r\tthree \t\r ' | perl -anE 'say scalar @F'
3
$ printf '     one \t\f\v two\t\r\tthree \t\r ' | perl -anE 'say "$F[1]."'
two.
```

Input field separator

You can use the `-F` command line option to specify a custom regexp field separator. Note that `-a` option implicitly sets `-n` and `-F` option implicitly sets `-n` and `-a` on newer versions of Perl. However, this book will always explicitly use these options.

```
$ # use ':' as input field separator
$ echo 'goal:amazing:whistle:kwality' | perl -F: -anE 'say "$F[0]\n$F[2]"'
goal
whistle

$ # use quotes to avoid clashes with shell special characters
$ echo 'one;two;three;four' | perl -F';' -anE 'say $F[2]'
three

$ echo 'load;err_msg--\ant,r2..not' | perl -F'\W+' -anE 'say $F[2]'
ant

$ echo 'hi.bye.hello' | perl -F'\.' -anE 'say $F[1]'
bye
```

You can also specify the regexp to `-F` option inside `//` delimiters as well as add `LIMIT` argument if needed.

```
$ # count number of vowels for each input line
$ # can also use: -F'(?i)[aeiou]'
$ printf 'COOL\nnice car\n' | perl -F'/[aeiou]/i' -anE 'say $#F'
2
3

$ # note that newline character is present as part of the last field content
$ echo 'goal:amazing:whistle:kwality' | perl -F'/:/,$_,2' -ane 'print $F[1]'
amazing:whistle:kwality
```

To get individual characters, you can use empty argument for the `-F` option.

```
$ echo 'apple' | perl -F -anE 'say $F[0]'
a
$ # -CS option will turn on UTF-8 for stdin/stdout/stderr streams
$ echo 'fox:αλεπού' | perl -CS -F -anE 'say @F[4..6]'
αλε
```

For more information about using `perl` with different encodings, see:

- perldoc: -C option
- unix.stackexchange: tr with unicode characters
- stackoverflow: Why does modern Perl avoid UTF-8 by default?

If the custom field separator with `-F` option doesn't affect the newline character, then the last element can contain the newline character.

```
$ # last element will not have newline character with default -a
$ # as leading/trailing whitespaces are trimmed with default split
$ echo 'cat dog' | perl -anE 'say "[$F[-1]]"'
[dog]

$ # last element will have newline character since field separator is ':'
$ echo 'cat:dog' | perl -F: -anE 'say "[$F[-1]]"'
[dog
]
$ # unless the input itself doesn't have newline character
$ printf 'cat:dog' | perl -F: -anE 'say "[$F[-1]]"'
[dog]
```

The newline character can also show up as the entire content of the last field.

```
$ # both leading and trailing whitespaces are trimmed
$ echo '   a b    c    ' | perl -anE 'say $#F'
2
$ # leading empty element won't be removed here
$ # and last element will have only newline character as its value
$ echo ':a:b:c:' | perl -F: -anE 'say $#F; say "[$F[-1]]"'
4
[
]
```

As mentioned before, the `-l` option is helpful if you wish to remove the newline character (more details will be discussed in Record separators chapter). A side effect of removing the newline character before applying `split` is that a trailing empty field will also get removed (you can explicitly call `split` function with `-1` as limit to prevent this).

```
$ # -l will remove the newline character
$ # -l will also cause 'print' to append the newline character
$ echo 'cat:dog' | perl -F: -lane 'print "[$F[-1]]"'
[dog]

$ # since newline character is chomped, last element is empty
$ # which is then removed due to default 'split' behavior
$ echo ':a:b:c:' | perl -F: -lane 'print scalar @F'
4
$ # explicit call to split with -1 as limit will preserve the empty element
$ echo ':a:b:c:' | perl -lne 'print scalar split/:/,$_,-1'
5
```

As per perldoc: -F option, "You can't use literal whitespace or NUL characters in the pattern." Here's some examples.

```
$ # only one element, field separator didn't match at all!!
$ echo 'pick eat rest laugh' | perl -F'/t /' -lane 'print $F[0]'
pick eat rest laugh
$ # number of splits is correct
$ # but the space character shouldn't be part of field here
$ echo 'pick eat rest laugh' | perl -F't ' -lane 'print $F[1]'
 res
$ # this gives the expected behavior
$ echo 'pick eat rest laugh' | perl -F't\x20' -lane 'print $F[1]'
res

$ # Error!!
$ echo 'pick eat rest laugh' | perl -F't[ ]' -lane 'print $F[1]'
Unmatched [ in regex; marked by <-- HERE in m/t[ <-- HERE /.
$ # no issues if 'split' is used explicitly
$ echo 'pick eat rest laugh' | perl -lne 'print((split /t[ ]/)[1])'
res
```

```
$ # example with NUL specified literally and as an escape sequence
$ printf 'a\0b\0c' | perl -F$'\0' -anE 'say join ",", @F' | cat -v
a,^@,b,^@,c
$ printf 'a\0b\0c' | perl -F'\0' -anE 'say join ",", @F' | cat -v
a,b,c
```

Output field separator

There are a few ways to affect the separator to be used while displaying multiple values.

Method 1: The value of `$,` special variable is used as the separator when multiple arguments (or list/array) are passed to `print` and `say` functions. `$,` could be remembered easily by noting that `,` is used to separate multiple arguments. Note that `-l` option is used in the examples below as a good practice even when not needed.

See perldoc: perlvar for alternate names of special variables if you use metacpan: English module. For example, `$OFS` or `$OUTPUT_FIELD_SEPARATOR` instead of `$,`

```
$ perl -lane 'BEGIN{$,=" "} print $F[0], $F[2]' table.txt
brown mat
blue mug
yellow window

$ s='Sample123string42with777numbers'
$ echo "$s" | perl -F'\d+' -lane 'BEGIN{$,=","} print @F'
Sample,string,with,numbers

$ # default value of $, is undef
$ echo 'table' | perl -F -lane 'print @F[0..2]'
tab
```

Method 2: By using the `join` function.

```
$ s='Sample123string42with777numbers'
$ echo "$s" | perl -F'\d+' -lane 'print join ",", @F'
Sample,string,with,numbers

$ s='goal:amazing:whistle:kwality'
$ echo "$s" | perl -F: -lane 'print join "-", @F[-1, 1, 0]'
kwality-amazing-goal
$ echo "$s" | perl -F: -lane 'print join "::", @F, 42'
goal::amazing::whistle::kwality::42
```

Method 3: You can also manually build the output string within double quotes. Or use `$"` to specify the field separator for an array value within double quotes. `$"` could be remembered easily by noting that interpolation happens within double quotes.

```
$ s='goal:amazing:whistle:kwality'
$ echo "$s" | perl -F: -lane 'print "$F[0] $F[2]"'
goal whistle

$ # default value of $" is space
$ echo "$s" | perl -F: -lane 'print "@F[0, 2]"'
goal whistle

$ echo "$s" | perl -F: -lane 'BEGIN{$"="-"} print "msg: @F[-1, 1, 0]"'
msg: kwality-amazing-goal
```

Changing number of fields

Manipulating `$#F` will change the number of fields for `@F` array.

```
$ s='goal:amazing:whistle:kwality'

$ # reducing fields
$ echo "$s" | perl -F: -lane '$#F=1; print join ",", @F'
goal,amazing

$ # increasing fields
$ echo "$s" | perl -F: -lane '$F[$#F+1]="sea"; print join ":", @F'
goal:amazing:whistle:kwality:sea

$ # empty fields will be created as needed
$ echo "$s" | perl -F: -lane '$F[7]="go"; print join ":", @F'
goal:amazing:whistle:kwality:::go
```

Assigning `$#F` to `-1` or lower will delete all the fields.

```
$ echo "1:2:3" | perl -F: -lane '$#F=-1; print "[@F]"'
[]
```

Here’s an example of adding a new field based on existing fields.

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67

$ # adds a new grade column based on marks in 3rd column
$ perl -anE 'BEGIN{$,="\t"; @g = qw(D C B A S)}
             say @F, $.==1 ? "Grade" : $g[$F[-1]/10 - 5]' marks.txt
Dept    Name    Marks   Grade
ECE     Raj     53      D
ECE     Joel    72      B
```

```
EEE     Moi     68      C
CSE     Surya   81      A
EEE     Tia     59      D
ECE     Om      92      S
CSE     Amy     67      C
```

Defining field contents instead of using split

The `-F` option uses the `split` function to get field values from input content. In contrast, using `/regexp/g` allows you to define what should the fields be made up of. Quoting from perldoc: Global matching

> In list context, `/g` returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regexp.

```
$ s='Sample123string42with777numbers'

$ # define fields to be one or more consecutive digits
$ # can also use: perl -nE 'say((/\d+/g)[1])'
$ echo "$s" | perl -nE '@f=/\d+/g; say $f[1]'
42

$ # define fields to be one or more consecutive alphabets
$ echo "$s" | perl -lne 'print join ",", /[a-z]+/ig'
Sample,string,with,numbers
```

Here's some examples to display results only if there's a match. Without the `if` conditions, you'll get empty lines for non-matching lines. Quoting from perldoc: The empty pattern //

> If the *PATTERN* evaluates to the empty string, the last successfully matched regular expression is used instead. In this case, only the `g` and `c` flags on the empty pattern are honored; the other flags are taken from the original pattern. If no match has previously succeeded, this will (silently) act instead as a genuine empty pattern (which will always match).

```
$ perl -nE 'say join "\n", //g if /\bm\w*\b/' table.txt
mat
mug

$ # /\bb\w*\b/ will come into play only if a word starting with 'h' isn't found
$ # so, first line matches 'hair' but not 'brown' or 'bread'
$ # other lines don't have words starting with 'h'
$ perl -nE 'say join "\n", //g if /\bh\w*\b/ || /\bb\w*\b/' table.txt
hair
blue
banana
```

As an alternate, you can use `while` loop with `g` flag. Quoting from perldoc: Global matching

In scalar context, successive invocations against a string will have `/g` jump from match to match, keeping track of position in the string as it goes along.

```
$ perl -nE 'say $& while /\bm\w*\b/g' table.txt
mat
mug

# note that this form isn't suited for priority extraction
$ perl -nE 'say $& while /\b[bh]\w*\b/g' table.txt
brown
bread
hair
blue
banana
```

A simple `split` fails for `csv` input where fields can contain embedded delimiter characters. For example, a field content `"fox,42"` when `,` is the delimiter.

```
$ s='eagle,"fox,42",bee,frog'
$ # simply using , as separator isn't sufficient
$ echo "$s" | perl -F, -lane 'print $F[1]'
"fox
```

While metacpan: Text::CSV module should be preferred for robust `csv` parsing, regexp is enough for simple formats.

```
$ echo "$s" | perl -lne 'print((/"[^"]+"|[^,]+/g)[1])'
"fox,42"
```

Fixed width processing

The `unpack` function is more than just a different way of string slicing. It supports various formats and pre-processing, see perldoc: unpack, perldoc: pack and perldoc: perlpacktut for details.

In the example below, `a` indicates arbitrary binary string. The optional number that follows indicates length of the field.

```
$ cat items.txt
apple   fig banana
50      10  200

$ # here field widths have been assigned such that
$ # extra spaces are placed at the end of each field
$ # $_ is the default input string for 'unpack' function
$ perl -lne 'print join ",", unpack "a8a4a6"' items.txt
apple   ,fig ,banana
50      ,10  ,200
$ perl -lne 'print((unpack "a8a4a6")[1])' items.txt
fig
10
```

You can specify characters to be ignored with `x` followed by optional length.

```
$ # first field is 5 characters
$ # then 3 characters are ignored and 3 characters for second field
$ # then 1 character is ignored and 6 characters for third field
$ perl -lne 'print join ",", unpack "a5x3a3xa6"' items.txt
apple,fig,banana
50   ,10 ,200
```

Using `*` will cause remaining characters of that particular format to be consumed. Here `Z` is used to process ASCII NUL separated string.

```
$ printf 'banana\x0050\x00' | perl -nE 'say join ":", unpack "Z*Z*"'
banana:50

$ # first field is 5 characters, then 3 characters are ignored
$ # all the remaining characters are assigned to second field
$ perl -lne 'print join ",", unpack "a5x3a*"' items.txt
apple,fig banana
50   ,10  200
```

Unpacking isn't always needed, string slicing using `substr` may suffice. See perldoc: substr for documentation.

```
$ # same as: perl -F -anE 'say @F[2..4]'
$ echo 'b 123 good' | perl -nE 'say substr $_,2,3'
123
$ echo 'b 123 good' | perl -ne 'print substr $_,6'
good

$ # replacing arbitrary slice
$ echo 'b 123 good' | perl -pe 'substr $_,2,3,"gleam"'
b gleam good
```

See also perldoc: Functions for fixed-length data or records.

Assorted field processing functions

Having seen command line options and features commonly used for field processing, this section will highlight some of the built-in functions. There's just too many to meaningfully cover them in all in detail, so consider this to be just a brief overview of features. See also perldoc: Perl Functions by Category.

First up, the `grep` function that allows you to select fields based on a condition. In scalar context, it returns number of fields that matched the given condition. See perldoc: grep for documentation. See also unix.stackexchange: create lists of words according to binary numbers.

```
$ s='goal:amazing:42:whistle:kwality:3.14'

$ # fields containing 'in' or 'it' or 'is'
$ echo "$s" | perl -F: -lane 'print join ":", grep {/i[nts]/} @F'
amazing:whistle:kwality
```

```
$ # number of fields NOT containing a digit character
$ echo "$s" | perl -F: -lane 'print scalar grep {!/\d/} @F'
4

$ s='hour hand band mat heated apple'
$ echo "$s" | perl -lane 'print join "\n", grep {!/^h/ && length()<4} @F'
mat

$ echo '20 711 -983 5 21' | perl -lane 'print join ":", grep {$_ > 20} @F'
711:21

$ # maximum of one field containing 'r'
$ perl -lane 'print if 1 >= grep {/r/} @F' table.txt
blue cake mug shirt -7
yellow banana window shoes 3.14
```

The `map` function transforms each element according to the logic passed to it. See perldoc: map for documentation.

```
$ s='goal:amazing:42:whistle:kwality:3.14'
$ echo "$s" | perl -F: -lane 'print join ":", map {uc} @F'
GOAL:AMAZING:42:WHISTLE:KWALITY:3.14
$ echo "$s" | perl -F: -lane 'print join ":", map {/^[gw]/ ? uc : $_} @F'
GOAL:amazing:42:WHISTLE:kwality:3.14

$ echo '23 756 -983 5' | perl -lane 'print join ":", map {$_ ** 2} @F'
529:571536:966289:25

$ echo 'AaBbCc' | perl -F -lane 'print join " ", map {ord} @F'
65 97 66 98 67 99
$ # for in-place modification of the input array
$ echo 'AaBbCc' | perl -F -lane 'map {$_ = ord} @F; print "@F"'
65 97 66 98 67 99

$ echo 'a b c' | perl -lane 'print join ",", map {qq/"$_"/} @F'
"a","b","c"
```

Here's an example with `grep` and `map` combined.

```
$ s='hour hand band mat heated pineapple'
$ echo "$s" | perl -lane 'print join "\n", map {y/ae/X/r} grep {/^h/} @F'
hour
hXnd
hXXtXd
$ # with 'grep' alone, provided the transformation doesn't affect the condition
$ # also, @F will be changed here, above map+grep code will not affect @F
$ echo "$s" | perl -lane 'print join "\n", grep {y/ae/X/; /^h/} @F'
hour
hXnd
hXXtXd
```

Here's some examples with `sort` and `reverse` functions for arrays and strings. See perldoc: sort and perldoc: reverse for documentation.

```
$ # sorting numbers
$ echo '23 756 -983 5' | perl -lane 'print join " ", sort {$a <=> $b} @F'
-983 5 23 756

$ s='floor bat to dubious four'
$ # default alphabetic sorting in ascending order
$ echo "$s" | perl -lane 'print join ":", sort @F'
bat:dubious:floor:four:to

$ # sort by length of the fields in ascending order
$ echo "$s" | perl -lane 'print join ":", sort {length($a) <=> length($b)} @F'
to:bat:four:floor:dubious
$ # descending order
$ echo "$s" | perl -lane 'print join ":", sort {length($b) <=> length($a)} @F'
dubious:floor:four:bat:to

$ # same as: perl -F -lane 'print sort {$b cmp $a} @F'
$ echo 'foobar' | perl -F -lane 'print reverse sort @F'
roofba
```

Here's an example with multiple sorting conditions. If the transformation applied for each field is expensive, using Schwartzian transform can provide a faster result. See also stackoverflow: multiple sorting conditions.

```
$ s='try a bad to good i teal by nice how'

$ # longer words first, ascending alphabetic order as tie-breaker
$ echo "$s" | perl -anE 'say join ":",
                    sort {length($b) <=> length($a) or $a cmp $b} @F'
good:nice:teal:bad:how:try:by:to:a:i

$ # using Schwartzian transform
$ echo "$s" | perl -anE 'say join ":", map {$_->[0]}
                         sort {$b->[1] <=> $a->[1] or $a->[0] cmp $b->[0]}
                         map {[$_, length($_)]} @F'
good:nice:teal:bad:how:try:by:to:a:i
```

Here's an example for sorting in descending order based on header column names.

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67

```

```
$ perl -lane '@i = sort {$F[$b] cmp $F[$a]} 0..$#F if $.==1;
              print join "\t", @F[@i]' marks.txt
Name    Marks   Dept
Raj     53      ECE
Joel    72      ECE
Moi     68      EEE
Surya   81      CSE
Tia     59      EEE
Om      92      ECE
Amy     67      CSE
```

See Using modules chapter for more field processing functions.

Summary

This chapter discussed various ways in which you can split (or define) the input into fields and manipulate them. Many more examples will be discussed in later chapters.

Exercises

a) Extract only the contents between `()` or `)(` from each input line. Assume that `()` characters will be present only once every line.

```
$ cat brackets.txt
foo blah blah(ice) 123 xyz$
(almond-pista) choco
yo )yoyo( yo

##### add your solution here
ice
almond-pista
yoyo
```

b) For the input file `scores.csv`, extract `Name` and `Physics` fields in the format shown below.

```
$ cat scores.csv
Name,Maths,Physics,Chemistry
Blue,67,46,99
Lin,78,83,80
Er,56,79,92
Cy,97,98,95
Ort,68,72,66
Ith,100,100,100

##### add your solution here
Name:Physics
Blue:46
Lin:83
```

```
Er:79
Cy:98
Ort:72
Ith:100
```

c) For the input file `scores.csv`, display names of those who've scored above `80` in Maths.

```
##### add your solution here
Cy
Ith
```

d) Display the number of word characters for the given inputs. Word definition here is same as used in regular expressions. Can you construct two different solutions as indicated below?

```
$ # solve using 's' operator
$ echo 'hi there' | ##### add your solution here
7

$ # solve without using substitution or transliteration operator
$ echo 'u-no;co%."(do_12:as' | ##### add your solution here
12
```

e) Construct a solution that works for both the given sample inputs and the corresponding output shown.

```
$ s1='1 "grape" and "mango" and "guava"'
$ s2='("a 1""d""c-2""b")'

$ echo "$s1" | ##### add your solution here
"grape","guava","mango"
$ echo "$s2" | ##### add your solution here
"a 1","b","c-2","d"
```

f) Display only the third and fifth characters from each input line.

```
$ printf 'restore\ncat one\ncricket' | ##### add your solution here
so
to
ik
```

g) Transform the given input file `fw.txt` to get the output as shown below. If second field is empty (i.e. contains only space characters), replace it with `NA`.

```
$ cat fw.txt
1.3  rs   90  0.134563
3.8           6
5.2  ye       8.2387
4.2  kt   32  45.1

##### add your solution here
1.3,rs,0.134563
3.8,NA,6
5.2,ye,8.2387
4.2,kt,45.1
```

h) For the input file `scores.csv`, display the header as well as any row which contains `b` or `t` (irrespective of case) in the first field.

```
##### add your solution here
Name,Maths,Physics,Chemistry
Blue,67,46,99
Ort,68,72,66
Ith,100,100,100
```

i) Extract all whole words that contains `42` but not at the edge of a word. Assume a word cannot contain `42` more than once.

```
$ s='hi42bye nice1423 bad42 cool_42a 42fake'
$ echo "$s" | ##### add your solution here
hi42bye
nice1423
cool_42a
```

j) For the input file `scores.csv`, add another column named `GP` which is calculated out of `100` by giving `50%` weightage to `Maths` and `25%` each for `Physics` and `Chemistry`.

```
##### add your solution here
Name,Maths,Physics,Chemistry,GP
Blue,67,46,99,69.75
Lin,78,83,80,79.75
Er,56,79,92,70.75
Cy,97,98,95,96.75
Ort,68,72,66,68.5
Ith,100,100,100,100.0
```

k) For the input file `mixed_fs.txt`, retain only first two fields from each input line. The input and output field separators should be space for first two lines and `,` for the rest of the lines.

```
$ cat mixed_fs.txt
rose lily jasmine tulip
pink blue white yellow
car,mat,ball,basket
light green,brown,black,purple

##### add your solution here
rose lily
pink blue
car,mat
light green,brown
```

l) For the given space separated numbers, filter only numbers in the range `20` to `1000` (inclusive).

```
$ s='20 -983 5 756 634223'

##### add your solution here
20 756
```

m) For the given input file `words.txt`, filter all lines containing characters in ascending and descending order.

```
$ cat words.txt
bot
art
are
boat
toe
flee
reed

$ # ascending order
##### add your solution here
bot
art

$ # descending order
##### add your solution here
toe
reed
```

n) For the given space separated words, extract the three longest words.

```
$ s='I bought two bananas and three mangoes'

$ echo "$s" | ##### add your solution here
bananas
mangoes
bought
```

o) Convert the contents of `split.txt` as shown below.

```
$ cat split.txt
apple,1:2:5,mango
wry,4,look
pencil,3:8,paper

##### add your solution here
apple,1,mango
apple,2,mango
apple,5,mango
wry,4,look
pencil,3,paper
pencil,8,paper
```

p) Generate string combinations as shown below for the given input string passed as an environment variable.

```
$ s='{x,y,z}{1,2,3}' ##### add your solution here
x1 x2 x3 y1 y2 y3 z1 z2 z3
```