

learnbyexample

# CLI text processing with GNU grep and ripgrep



✓ **200+ examples**

✓ **50+ exercises**

**Sundeep Agarwal**

# Table of contents

<b>Preface</b>	<b>4</b>
Prerequisites . . . . .	4
Conventions . . . . .	4
Acknowledgements . . . . .	4
Feedback and Errata . . . . .	5
Author info . . . . .	5
License . . . . .	5
Book version . . . . .	6
<b>Introduction</b>	<b>7</b>
Installation . . . . .	7
Options overview . . . . .	8
<b>Frequently used options</b>	<b>10</b>
Basic string search . . . . .	10
Fixed string search . . . . .	11
Case insensitive search . . . . .	12
Invert matching lines . . . . .	12
Line number and count . . . . .	12
Limiting output lines . . . . .	13
Multiple search strings . . . . .	13
Get filename instead of matching lines . . . . .	14
Filename prefix for matching lines . . . . .	15
Quickfix . . . . .	15
Colored output . . . . .	16
Match whole word . . . . .	17
Match whole line . . . . .	17
Comparing lines between files . . . . .	18
Extract only matching portion . . . . .	18
Summary . . . . .	19
Interactive exercises . . . . .	19
Exercises . . . . .	20
<b>BRE/ERE Regular Expressions</b>	<b>23</b>
Line Anchors . . . . .	23
Word Anchors . . . . .	24
Opposite Word Anchor . . . . .	25
Alternation . . . . .	25
Alternation precedence . . . . .	27
Grouping . . . . .	27
Escaping metacharacters . . . . .	28
Matching characters like tabs . . . . .	28
The dot metacharacter . . . . .	29
Quantifiers . . . . .	30
Conditional AND . . . . .	32
Longest match wins . . . . .	32
Character classes . . . . .	33
Character class metacharacters . . . . .	34

Escape sequence sets . . . . .	36
Named character sets . . . . .	36
Matching character class metacharacters literally . . . . .	37
Backreferences . . . . .	38
Known Bugs . . . . .	39
Summary . . . . .	40
Exercises . . . . .	40

# Preface

You are likely to be familiar with using a search dialog (usually invoked with the `Ctrl+F` shortcut) to locate the occurrences of a particular string. Graphical User Interface (GUI) tools such as a text editor, word processor, web browser and programming IDE usually support such a search feature. The `grep` command is a versatile and feature-rich version of that search functionality usable from the command line. An important feature that GUI applications may lack is **regular expressions**, a mini-programming language to precisely define a matching criteria.

Modern requirements have given rise to tools like `ripgrep` that provide out-of-box features such as recursive search while respecting the ignore rules of a version controlled directory.

This book heavily leans on examples to present features one by one. In addition to command options, regular expressions will also be discussed in detail. It is recommended that you manually type each example. Make an effort to understand the sample input as well as the solution presented and check if the output changes (or not!) when you alter some part of the input and the command. As an analogy, consider learning to drive a car — no matter how much you read about them or listen to explanations, you'd need practical experience to become proficient.

## Prerequisites

You should be familiar with command line usage in a Unix-like environment. You should also be comfortable with concepts like file redirection and command pipelines.

You are also expected to get comfortable with reading manuals, searching online, visiting external links provided for further reading, tinkering with illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

If you are new to the world of the command line, check out my [Computing from the Command Line](#) ebook and curated resources on [Linux CLI and Shell scripting](#) before starting this book.

## Conventions

- Code snippets are copy pasted from the Bash shell (version **5.0.17**) for **GNU grep 3.10** and **ripgrep 13.0.0**. Such snippets have been modified for presentation purposes — some commands are preceded by comments to provide context and explanations, blank lines have been added to improve readability and so on.
- Unless otherwise noted, all examples and explanations are meant for **ASCII** input.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The [learn\\_gnugrep\\_ripgrep repo](#) has all the [example files and scripts](#) used in the book. The repo also includes [code snippets](#) and [exercises](#) used in the book. Solutions file is also provided. If you are not familiar with the `git` command, click the **Code** button on the webpage to get the files.

## Acknowledgements

- [GNU grep documentation](#) — manual and examples
- [ripgrep](#) — user guide and examples

- [stackoverflow](#) and [unix.stackexchange](#) — for getting answers to pertinent questions on `grep` and related commands
- [tex.stackexchange](#) — for help on `pandoc` and `tex` related questions
- [/r/commandline/](#), [/r/linux4noobs/](#), [/r/linuxquestions/](#) and [/r/linux/](#) — helpful forums
- [canva](#) — cover image
- [oxipng](#), [pngquant](#) and [svgcleaner](#) — optimizing images
- [Warning](#) and [Info](#) icons by [Amada44](#) under public domain
- [carbon](#) — creating terminal screenshots with highlighted text
- [Andrew Gallant](#) (author of `ripgrep`), [mikeblas](#) and [Pound-Hash](#) for critical feedback

Special thanks to all my friends and online acquaintances for their help, support and encouragement, especially during difficult times.

## Feedback and Errata

I would highly appreciate it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: [https://github.com/learnbyexample/learn\\_gnugrep\\_ripgrep/issues](https://github.com/learnbyexample/learn_gnugrep_ripgrep/issues)
- E-mail: [learnbyexample.net@gmail.com](mailto:learnbyexample.net@gmail.com)
- Twitter: [https://twitter.com/learn\\_byexample](https://twitter.com/learn_byexample)

## Author info

Sundee Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at <https://github.com/learnbyexample>.

**List of books:** <https://learnbyexample.github.io/books/>

## License

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Code snippets are available under [MIT License](#).

Resources mentioned in Acknowledgements section above are available under original licenses.

## Book version

2.0

See [Version\\_changes.md](#) to track changes across book versions.

# Introduction

Quoting from [wikipedia](#):

`grep` is a command-line utility for searching plain-text data sets for lines that match a regular expression. Its name comes from the `ed` command `g/re/p` (*g*lobal / *r*egular *e*xpression search / and *p*rint), which has the same effect.

Use of `grep` has become so ubiquitous that it has found its way into the [Oxford dictionary](#) as well. As part of everyday computer usage, the need to search comes up often. It could be finding the right emoji by name on social media, searching your browser bookmarks, locating a particular function in a programming file and so on. Some of these tools have options for refining a search further, like controlling case sensitivity, restricting matches to whole words, using regular expressions, etc.

`grep` provides all of the above features and much more when it comes to searching and extracting content from text files. After getting used to `grep`, the search features provided by GUI programs feel slower and inadequate.

## Installation

If you are on a Unix-like system, you will most likely have some version of `grep` already installed. This book is primarily about `GNU grep` and also has a chapter on `ripgrep`. As there are syntax and feature differences between various implementations, make sure to have these particular commands to follow along the examples presented in this book.

`GNU grep` is part of the [text creation and manipulation](#) tools and comes by default on GNU/Linux distributions. To install a particular version, visit [gnu: grep software](#). See also [release notes](#) for an overview of changes between versions and [bug list](#) if you think some command isn't working as expected.

Sample instructions for compiling the latest version are shown below. You might need to install a PCRE library first, for example `sudo apt install libpcre2-dev`.

```
$ wget https://ftp.gnu.org/gnu/grep/grep-3.10.tar.xz
$ tar -xf grep-3.10.tar.xz
$ cd grep-3.10/
# see https://askubuntu.com/q/237576 if you get compiler not found error
$ ./configure
$ make
$ sudo make install

$ grep -V | head -n1
grep (GNU grep) 3.10
```

If you are not using a Linux distribution, you may be able to access `GNU grep` using an option below:

- [Git for Windows](#) — provides a Bash emulation used to run Git from the command line
- [Windows Subsystem for Linux](#) — compatibility layer for running Linux binary executables natively on Windows
- [brew](#) — Package Manager for macOS (or Linux)

## Options overview

It is always good to know where to find documentation. From the command line, you can use `man grep` for a short manual and `info grep` for the full documentation. I prefer using the [online gnu grep manual](#), which feels much easier to use and navigate.

```
$ man grep
NAME
    grep - print lines that match patterns

SYNOPSIS
    grep [OPTION...] PATTERNS [FILE...]
    grep [OPTION...] -e PATTERNS ... [FILE...]
    grep [OPTION...] -f PATTERN_FILE ... [FILE...]

DESCRIPTION
    grep searches for PATTERNS in each FILE. PATTERNS is one or more
    patterns separated by newline characters, and grep prints each
    line that matches a pattern. Typically PATTERNS should be quoted
    when grep is used in a shell command.

    A FILE of "-" stands for standard input. If no FILE is given,
    recursive searches examine the working directory, and
    nonrecursive searches read standard input.
```

For a quick overview of all the available options, use `grep --help` from the command line. These are shown below in table format:

### Regex selection:

Option	Description
-E, --extended-regexp	PATTERNS are extended regular expressions
-F, --fixed-strings	PATTERNS are strings
-G, --basic-regexp	PATTERNS are basic regular expressions
-P, --perl-regexp	PATTERNS are Perl regular expressions
-e, --regexp=PATTERNS	use PATTERNS for matching
-f, --file=FILE	take PATTERNS from FILE
-i, --ignore-case	ignore case distinctions in patterns and data
--no-ignore-case	do not ignore case distinctions (default)
-w, --word-regexp	match only whole words
-x, --line-regexp	match only whole lines
-z, --null-data	a data line ends in 0 byte, not newline

### Miscellaneous:

Option	Description
-s, --no-messages	suppress error messages
-v, --invert-match	select non-matching lines
-V, --version	display version information and exit
--help	display this help text and exit



**Output control:**

Option	Description
-m, --max-count=NUM	stop after NUM selected lines
-b, --byte-offset	print the byte offset with output lines
-n, --line-number	print line number with output lines
--line-buffered	flush output on every line
-H, --with-filename	print file name with output lines
-h, --no-filename	suppress the file name prefix on output
--label=LABEL	use LABEL as the standard input file name prefix
-o, --only-matching	show only nonempty parts of lines that match
-q, --quiet, --silent	suppress all normal output
--binary-files=TYPE	assume that binary files are TYPE; TYPE is 'binary', 'text', or 'without-match'
-a, --text	equivalent to --binary-files=text
-I	equivalent to --binary-files=without-match
-d, --directories=ACTION	how to handle directories; ACTION is 'read', 'recurse', or 'skip'
-D, --devices=ACTION	how to handle devices, FIFOs and sockets; ACTION is 'read' or 'skip'
-r, --recursive	like --directories=recurse
-R, --dereference-recursive	likewise, but follow all symlinks
--include=GLOB	search only files that match GLOB (a file pattern)
--exclude=GLOB	skip files that match GLOB
--exclude-from=FILE	skip files that match any file pattern from FILE
--exclude-dir=GLOB	skip directories that match GLOB
-L, --files-without-match	print only names of FILES with no selected lines
-l, --files-with-matches	print only names of FILES with selected lines
-c, --count	print only a count of selected lines per FILE
-T, --initial-tab	make tabs line up (if needed)
-Z, --null	print 0 byte after FILE name

**Context control:**

Option	Description
-B, --before-context=NUM	print NUM lines of leading context
-A, --after-context=NUM	print NUM lines of trailing context
-C, --context=NUM	print NUM lines of output context
-NUM	same as --context=NUM
--group-separator=SEP	print SEP on line between matches with context
--no-group-separator	do not print separator for matches with context
--color[=WHEN], --colour[=WHEN]	use markers to highlight the matching strings; WHEN is 'always', 'never', or 'auto'
-U, --binary	do not strip CR characters at EOL (MSDOS/Windows)

## Frequently used options

This chapter will cover many of the options provided by `GNU grep`. Regular expressions will be covered later, so the examples in this chapter will only use literal strings as search patterns. Literal (fixed string) matching refers to exact string comparison, so no special meaning is assigned for any of the search characters.



The `example_files` directory has all the files used in the examples.

### Basic string search

By default, `grep` will print all the input *lines* that match the given search patterns. The newline character `\n` is the line separator by default. This section will show you how to filter lines matching a given search string using `grep`. Consider this sample input file:

```
$ cat ip.txt
it is a warm and cozy day
listen to what I say
go play in the park
come back before the sky turns dark

There are so many delights to cherish
Apple, Banana and Cherry
Bread, Butter and Jelly
Try them all before you perish
```

To filter desired lines, invoke the `grep` command, pass the search string and then specify one or more filenames that have to be searched. As a good practice, always use single quotes around the search string. Examples requiring shell interpretation will be discussed later.

```
$ grep 'play' ip.txt
go play in the park

$ grep 'y t' ip.txt
come back before the sky turns dark
Try them all before you perish
```

`grep` will perform the search on `stdin` data if there are no file arguments or if `-` is used as a filename.

```
$ printf 'apple\nbanana\nmango\nfig\n' | grep 'an'
banana
mango

$ printf 'apple\nbanana\nmango\nfig\n' | grep 'an' -
banana
mango
```

Here's an example where `grep` reads user written `stdin` data and the filtered output is redirected to a file.

```
# press Ctrl+d after the line containing 'histogram'
$ grep 'is' > op.txt
hi there
this is a sample line
have a nice day
histogram

$ cat op.txt
this is a sample line
histogram

$ rm op.txt
```



If your input file has `\r\n` (carriage return and newline characters) as the line ending, convert the input file to Unix-style before processing. See [stackoverflow: Why does my tool output overwrite itself and how do I fix it?](#) for a detailed discussion and mitigation methods.

```
# Unix style
$ printf '42\n' | file -
/dev/stdin: ASCII text

# DOS style
$ printf '42\r\n' | file -
/dev/stdin: ASCII text, with CRLF line terminators
```

## Fixed string search

The search string (pattern) is treated as a Basic Regular Expression (BRE) by default. But regular expressions is a topic for the next chapter. For now, use the `-F` option to indicate that the patterns should be matched literally.

```
# oops, why did it not match?
$ echo 'int a[5]' | grep 'a[5]'
# where did that error come from??
$ echo 'int a[5]' | grep 'a['
grep: Invalid regular expression
# what is going on???
$ echo 'int a[5]' | grep 'a[5]'
grep: Unmatched [, [^, [:, [., or [=

# use the -F option to match strings literally
$ echo 'int a[5]' | grep -F 'a[5]'
int a[5]
```



If the search string doesn't have any regular expression metacharacters, GNU `grep` will try a literal search even if the `-F` option isn't used.

## Case insensitive search

Sometimes, you don't know if a log file contains case variable search terms, such as `error` , `Error` , or `ERROR` . In such cases, you can use the `-i` option to ignore case.

```
$ grep -i 'the' ip.txt
go play in the park
come back before the sky turns dark
There are so many delights to cherish
Try them all before you perish

$ printf 'Cat\ncOnCaT\ncut\n' | grep -i 'cat'
Cat
cOnCaT
```

## Invert matching lines

Use the `-v` option to get lines other than those matching the search term.

```
$ seq 4 | grep -v '3'
1
2
4

$ printf 'goal\ncrate\near\npit' | grep -v 'at'
goal
pit
```



Text processing often involves negating a logic to arrive at a solution or to make it simpler. Look out for opposite pairs like `-l -L` , `-h -H` , negative logic in regular expressions and so on in the examples to follow.

## Line number and count

The `-n` option will prefix line numbers to matching results, using a colon character as the separator. This is useful to quickly locate matching lines for further processing.

```
$ grep -n 'to' ip.txt
2:listen to what I say
6:There are so many delights to cherish

$ printf 'great\nneat\nuser' | grep -n 'eat'
1:great
2:neat
```

Having to count the total number of matching lines comes up often. Somehow piping `grep` output to the `wc` command is prevalent instead of simply using the `-c` option.

```
# number of lines matching the pattern
$ grep -c 'is' ip.txt
4
```

```
# number of lines NOT matching the pattern
$ printf 'goal\nrate\neat\npit' | grep -vc 'g'
3
```

When multiple input files are passed, the count is displayed for each file separately. Use `cat` if you need a combined count.

```
# here - represents the stdin data
$ printf 'this\nis\ncool\n' | grep -c 'is' ip.txt -
ip.txt:4
(standard input):2

# useful application of the cat command
$ cat <(printf 'this\nis\ncool\n') ip.txt | grep -c 'is'
6
```



The output given by the `-c` option is the total number of **lines** matching the given patterns, not the total number of matches. Use the `-o` option, and pipe the output to `wc -l` to count every occurrence (example shown later).

## Limiting output lines

Sometimes, there are too many results, in which case you could pipe the output to a **pager** tool like `less`. Or use the `-m` option to limit how many matching lines should be displayed for each input file. `grep` will stop processing an input file as soon as the condition specified by `-m` is satisfied. Note that just like the `-c` option, `-m` works by line count and not based on the total number of matches.

```
$ grep -m2 'is' ip.txt
it is a warm and cozy day
listen to what I say

$ seq 1000 | grep -m4 '2'
2
12
20
21
```

## Multiple search strings

The `-e` option can be used to specify multiple search strings from the command line. This is similar to conditional OR boolean logic.

```
# search for 'what' or 'But'
$ grep -e 'what' -e 'But' ip.txt
listen to what I say
Bread, Butter and Jelly
```

If you have a huge list of strings to search, save them in a file, one search string per line. Make sure there are no empty lines. Then use the `-f` option to specify a file as the source of search

strings. You can use this option multiple times and also add more patterns from the command line using the `-e` option. Also, add the `-F` option when searching for literal matches. It is easy to miss regular expression metacharacters in a big list of terms.

```
$ cat search.txt
say
you

$ grep -Ff search.txt ip.txt
listen to what I say
Try them all before you perish

# example with both -f and -e options
$ grep -Ff search.txt -e 'it' -e 'are' ip.txt
it is a warm and cozy day
listen to what I say
There are so many delights to cherish
Try them all before you perish
```

To find lines matching more than one search term, you'd need to either resort to using regular expressions (covered later) or workaround by using shell pipes. This is similar to conditional AND boolean logic.

```
# match lines containing both 'is' and 'to' in any order
# same as: grep 'to' ip.txt | grep 'is'
$ grep 'is' ip.txt | grep 'to'
listen to what I say
There are so many delights to cherish
```

## Get filename instead of matching lines

Often, you just want a list of filenames that match the search patterns. The output might get saved for future reference or passed to another command like `sed`, `awk`, `perl`, `sort`, etc for further processing. Some of these commands can handle search by themselves, but `grep` is a fast and specialized tool for searching and using shell pipes can improve performance if parallel processing is available. Similar to the `-m` option, `grep` will stop processing the input file as soon as the given condition is satisfied.

- `-l` will list files matching the pattern
- `-L` will list files NOT matching the pattern

Here are some examples:

```
# list filename if it contains 'are'
$ grep -l 'are' ip.txt search.txt
ip.txt
# no output because no match was found
$ grep -l 'xyz' ip.txt search.txt
# list filename if it contains 'say'
$ grep -l 'say' ip.txt search.txt
ip.txt
search.txt
```

```
# list filename if it does NOT contain 'xyz'
$ grep -L 'xyz' ip.txt search.txt
ip.txt
search.txt
# list filename if it does NOT contain 'are'
$ grep -L 'are' ip.txt search.txt
search.txt
```

## Filename prefix for matching lines

If there are multiple input files, `grep` will automatically prefix the filename when displaying the matching lines. You can also control whether or not to add the prefix using the following options:

- `-h` option will prevent filename prefix in the output (default for single input file)
- `-H` option will always show filename prefix (default for multiple input files)

```
# -h is on by default for single input file
$ grep 'say' ip.txt
listen to what I say
# using -h to suppress filename prefix for multiple input files
$ printf 'say\nyou\n' | grep -h 'say' - ip.txt
say
listen to what I say

# -H is on by default for multiple input files
$ printf 'say\nyou\n' | grep 'say' - ip.txt
(standard input):say
ip.txt:listen to what I say
# use -H to always show filename prefix
# instead of -H, you can also provide /dev/null as an additional input file
$ grep -H 'say' ip.txt
ip.txt:listen to what I say
```

## Quickfix

The `vim` editor has a quickfix option `-q` that makes it easy to edit the matching lines from `grep`'s output. Make sure that the output has both line numbers and filename prefixes.

```
# -H ensures filename prefix and -n provides line numbers
$ grep -Hn 'say' ip.txt search.txt
ip.txt:2:listen to what I say
search.txt:1:say

# use :cn and :cp to navigate to next/previous occurrences
# command-line area at the bottom will show number of matches and filenames
# you can also save the grep output and pass that filename instead of <()
$ vim -q <(grep -Hn 'say' ip.txt search.txt)
```

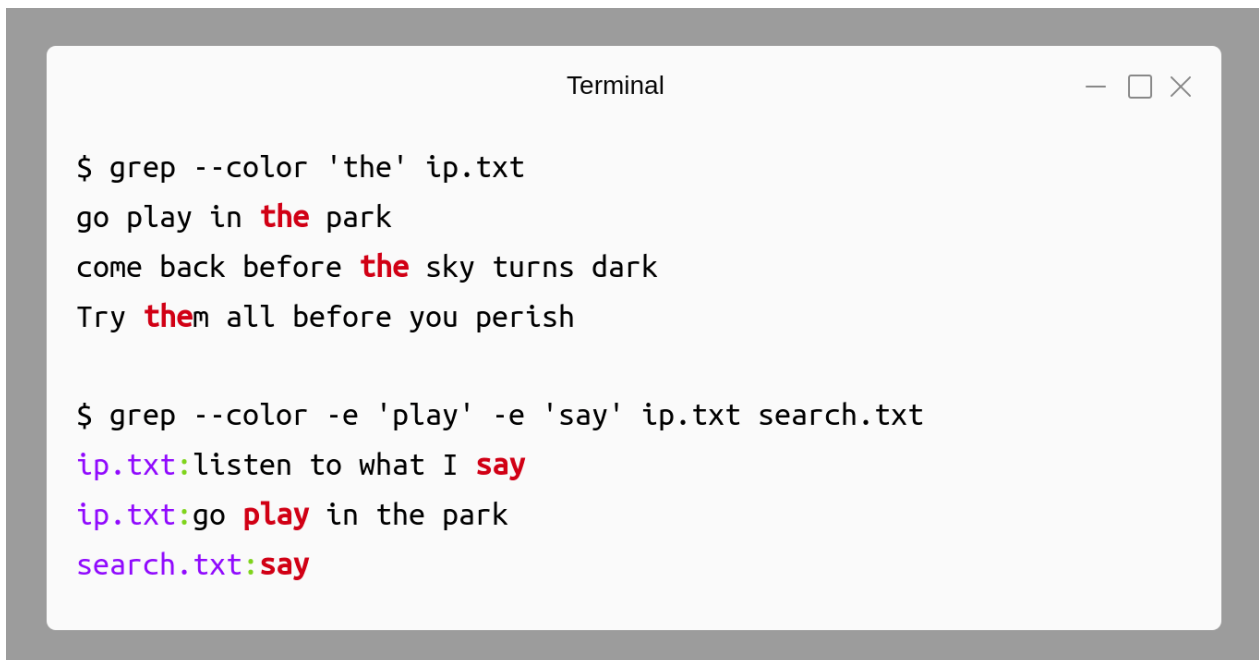
## Colored output

When working from the terminal, having the `--color` option enabled makes it easier to spot the matching portions in the output. Especially useful when you are experimenting to find the correct regular expression. Modern terminals will usually have color support, see [unix.stackexchange: How to check if bash can print colors?](https://unix.stackexchange.com/questions/110382/how-to-check-if-bash-can-print-colors/) for details.

The `--color` (or `--colour`) option will highlight matching patterns, line numbers, filenames, etc. There are three different settings:

- `auto` will result in color highlighting when results are displayed on terminal, but not when the output is redirected to another command, file, etc. This is the default setting
- `always` will result in color highlighting when results are displayed on terminal as well as when the output is redirected to another command, file, etc
- `never` explicitly disables color highlighting

Here are couple of examples with the `--color` option enabled (default is `auto`).

A terminal window titled "Terminal" with standard window controls (minimize, maximize, close). It shows two examples of the grep command with color highlighting. The first example shows the word "the" highlighted in red in three lines of text. The second example shows multiple matches highlighted in different colors: "say" in red, "play" in red, and "listen" in purple.

```
Terminal

$ grep --color 'the' ip.txt
go play in the park
come back before the sky turns dark
Try them all before you perish

$ grep --color -e 'play' -e 'say' ip.txt search.txt
ip.txt:listen to what I say
ip.txt:go play in the park
search.txt:say
```

It is typical to alias both the `ls` and `grep` commands to include `--color=auto`.

```
# aliases are usually saved in ~/.bashrc or ~/.bash_aliases
$ alias ls='ls --color=auto'
$ alias grep='grep --color=auto'
```

Using `--color=always` is handy if you want to retain color information even when the output is redirected. For example, piping the results to the `less` command.

```
$ grep --color=always -i 'the' ip.txt | less -R
```

The below image will help you understand the difference between the `auto` and `always` features. In the first case, `is` gets highlighted even after piping, while in the second case `is` loses the color information. In practice, `always` is rarely used as it provides extra information to matching lines, which could cause undesirable results when processed.



```
Terminal

$ grep --color=always 'is' ip.txt | grep --color 'to'
listen to what I say
There are so many delights to cherish

$ grep --color=auto 'is' ip.txt | grep --color 'to'
listen to what I say
There are so many delights to cherish
```

## Match whole word

A word character is any alphabet (irrespective of case), digit and the underscore character. You might wonder why there are digits and underscores as well, why not only alphabets? This comes from variable and function naming conventions — typically alphabets, digits and underscores are allowed. So, the definition is more programming oriented than natural language. The `-w` option will ensure that given patterns are not surrounded by other word characters. For example, this helps to distinguish `par` from `spar`, `park`, `apart`, `par2`, `_par`, etc.

```
# this matches 'par' anywhere in the line
$ printf 'par value\nheir apparent\n' | grep 'par'
par value
heir apparent
# this matches 'par' only as a whole word
$ printf 'par value\nheir apparent\n' | grep -w 'par'
par value
```



The `-w` option behaves a bit differently than word boundaries in regular expressions. See the [Word boundary differences](#) section for details.

## Match whole line

Another useful option is `-x`, which will display a line only if the entire line satisfies the given pattern.

```
# this matches 'my book' anywhere in the line
$ printf 'see my book list\nmy book\n' | grep 'my book'
see my book list
my book
# this matches 'my book' only as a whole line
$ printf 'see my book list\nmy book\n' | grep -x 'my book'
my book
```

```
$ grep 'say' ip.txt search.txt
ip.txt:listen to what I say
search.txt:say
$ grep -x 'say' ip.txt search.txt
search.txt:say

# count empty lines, won't work for files with DOS style line endings
$ grep -cx '' ip.txt
1
```

## Comparing lines between files

The `-f` and `-x` options can be combined to get common lines between two files or the difference when `-v` is used as well. If you want to match the lines literally, it is advised to use the `-F` option as well, because you might not know if there are regular expression metacharacters present in the input files or not.

```
$ printf 'teal\nlight blue\nbrown\nyellow\n' > colors_1
$ printf 'blue\nblack\ndark green\nyellow\n' > colors_2

# common lines between two files
$ grep -Fxf colors_1 colors_2
yellow

# lines present in colors_2 but not in colors_1
$ grep -Fvxf colors_1 colors_2
blue
black
dark green

# lines present in colors_1 but not in colors_2
$ grep -Fvxf colors_2 colors_1
teal
light blue
brown
```

See also [stackoverflow: Fastest way to find lines of a text file from another larger text file](#) — go through all the answers.

## Extract only matching portion

If the total number of matches is required, use the `-o` option to display only the matching portions (one per line), and then use `wc` to count them. This option is more commonly used with regular expressions.

```
$ grep -oi 'the' ip.txt
the
the
The
the
```

```
# -c only gives count of matching lines
$ grep -c 'an' ip.txt
4
# use -o to get each match on a separate line
$ grep -o 'an' ip.txt | wc -l
6
```

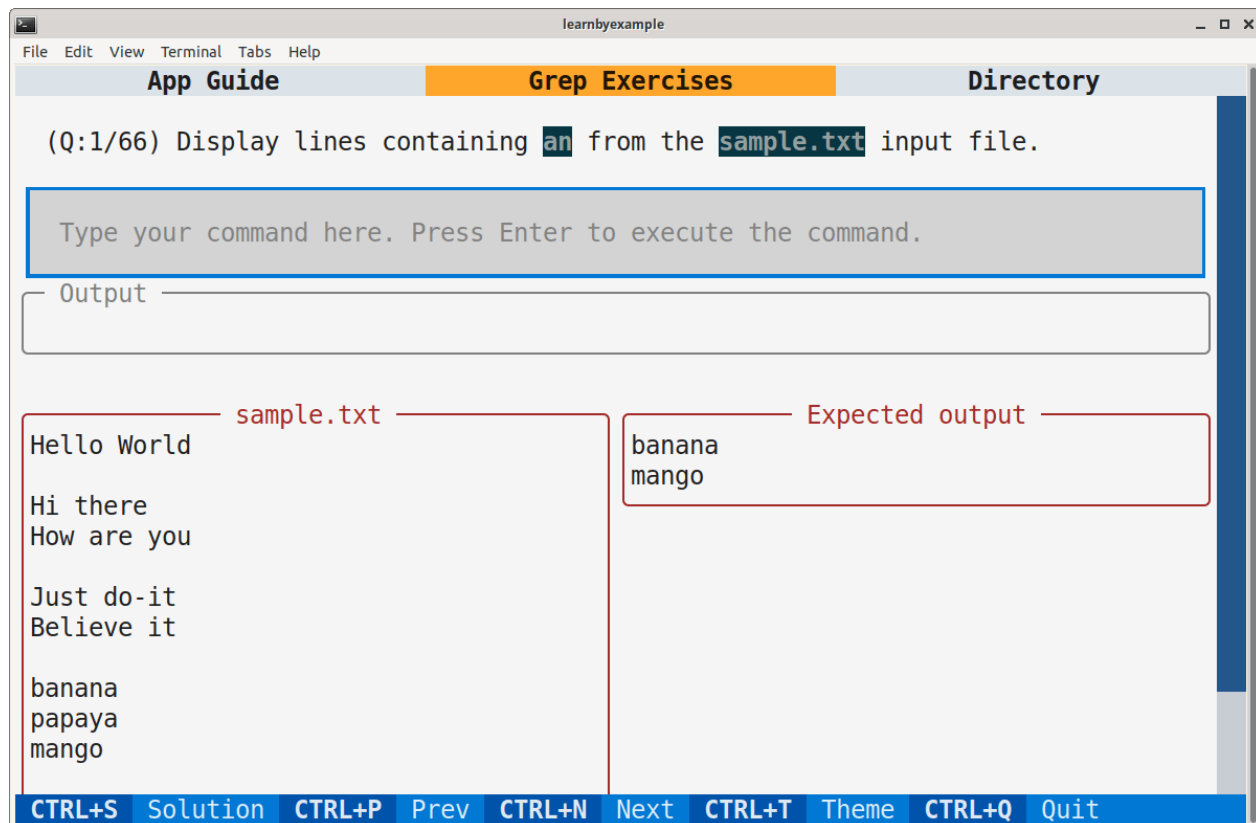
## Summary

In my initial years of CLI usage as a VLSI engineer, I knew only some of the options listed in this chapter. Didn't even know about the `--color` option. I've come across comments about not knowing the `-c` option in online forums. These are some of the reasons why I'd advise going through the list of all the options if you use a command frequently. Bonus points for maintaining a cheatsheet of example usage for future reference, passing on to your colleagues, etc.

## Interactive exercises

I wrote a TUI app to help you solve some of the exercises from this book interactively. See [GrepExercises](#) repo for installation steps and [app\\_guide.md](#) for instructions on using this app.

Here's a sample screenshot:



## Exercises



All the exercises are also collated together in one place at [Exercises.md](#). For solutions, see [Exercise\\_solutions.md](#).



The [exercises](#) directory has all the files used in this section.

1) Display lines containing `an` from the `sample.txt` input file.

```
##### add your solution here
banana
mango
```

2) Display lines containing `do` as a whole word from the `sample.txt` input file.

```
##### add your solution here
Just do-it
```

3) Display lines from `sample.txt` that satisfy both of these conditions:

- `he` matched irrespective of case
- either `World` or `Hi` matched case sensitively

```
##### add your solution here
Hello World
Hi there
```

4) Display lines from `code.txt` containing `fruit[0]` literally.

```
##### add your solution here
fruit[0] = 'apple'
```

5) Display only the first two matching lines containing `t` from the `sample.txt` input file.

```
##### add your solution here
Hi there
Just do-it
```

6) Display only the first three matching lines that do *not* contain `he` from the `sample.txt` input file.

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

How are you
```

7) Display lines from `sample.txt` that contain `do` along with line number prefix.

```
##### add your solution here
6:Just do-it
13:Much ado about nothing
```

8) For the input file `sample.txt`, count the number of times the string `he` is present, irrespective of case.

```
##### add your solution here
5
```

**9)** For the input file `sample.txt` , count the number of empty lines.

```
##### add your solution here
4
```

**10)** For the input files `sample.txt` and `code.txt` , display matching lines based on the search terms (one per line) present in the `terms.txt` file. Results should be prefixed with the corresponding input filename.

```
$ cat terms.txt
are
not
go
fruit[0]

##### add your solution here
sample.txt:How are you
sample.txt:mango
sample.txt:Much ado about nothing
sample.txt:Adios amigo
code.txt:fruit[0] = 'apple'
```

**11)** For the input file `sample.txt` , display lines containing `amigo` prefixed by the input filename as well as the line number.

```
##### add your solution here
sample.txt:15:Adios amigo
```

**12)** For the input files `sample.txt` and `code.txt` , display only the filename if it contains `apple` .

```
##### add your solution here
code.txt
```

**13)** For the input files `sample.txt` and `code.txt` , display only whole matching lines based on the search terms (one per line) present in the `lines.txt` file. Results should be prefixed with the corresponding input filename as well as the line number.

```
$ cat lines.txt
banana
fruit = []

##### add your solution here
sample.txt:9:banana
code.txt:1:fruit = []
```

**14)** For the input files `sample.txt` and `code.txt` , count the number of lines that do *not* match any of the search terms (one per line) present in the `terms.txt` file.

```
##### add your solution here
sample.txt:11
```

```
code.txt:3
```

**15)** Count the total number of lines containing `banana` in the input files `sample.txt` and `code.txt` .

```
##### add your solution here
2
```

**16)** Which two conditions are necessary for the output of the `grep` command to be suitable for the `vim -q` quickfix mode?

**17)** What's the default setting for the `--color` option? Give an example where the `always` setting would be useful.

**18)** The command shown below tries to get the number of empty lines, but apparently shows the wrong result, why?

```
$ grep -cx '' dos.txt
0
```

# BRE/ERE Regular Expressions

This chapter covers Basic and Extended Regular Expressions as implemented in `GNU grep`. Unless otherwise indicated, examples and descriptions will assume ASCII input. `GNU grep` also supports Perl Compatible Regular Expressions, which will be discussed in a [later chapter](#).

By default, `grep` treats the search pattern as Basic Regular Expression (BRE). Here are the various options available to choose a particular flavor:

- `-G` option can be used to specify explicitly that BRE is needed
- `-E` option will enable Extended Regular Expression (ERE)
  - in `GNU grep`, BRE and ERE only differ in how metacharacters are specified, no difference in features
- `-F` option will cause the search patterns to be treated literally
- `-P` if available, this option will enable Perl Compatible Regular Expression (PCRE)



The [example\\_files](#) directory has all the files used in the examples.



See [grep manual: Problematic Regular Expressions](#) if you are working on portable scripts. See also [POSIX specification for BRE and ERE](#).

## Line Anchors

Instead of matching anywhere in the line, restrictions can be specified. For now, you'll see the ones that are already part of BRE/ERE. In later sections and chapters, you'll get to know how to define your own rules for restriction. These restrictions are made possible by assigning special meaning to certain characters and escape sequences.

The characters with special meaning are known as **metacharacters** in regular expressions parlance. In case you need to match those characters literally, you need to escape them with a `\` (discussed in the [Escaping metacharacters](#) section).

There are two line anchors:

- `^` metacharacter restricts the matching to the start of the line
- `$` metacharacter restricts the matching to the end of the line

Here are some examples:

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

# lines starting with 's'
$ grep '^s' anchors.txt
sub par
spar
```

```
# lines ending with 'rt'
$ grep 'rt$' anchors.txt
apparent effort
cart part tart mart
```

You can combine these two anchors to match only whole lines. Or, use the `-x` option.

```
$ printf 'spared no one\npar\nspar\ndare' | grep '^par$'
par
$ printf 'spared no one\npar\nspar\ndare' | grep -x 'par'
par
```

## Word Anchors

The second type of restriction is word anchors. A word character is any alphabet (irrespective of case), digit and the underscore character. This is similar to using `-w` option, with added flexibility of using word anchor only at the start or end of a word.

The escape sequence `\b` denotes a word boundary. This works for both the start of word and the end of word anchoring. Start of word means either the character prior to the word is a non-word character or there is no character (start of line). Similarly, end of word means the character after the word is a non-word character or no character (end of line). This implies that you cannot have word boundaries without a word character. Here are some examples:

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

# match words starting with 'par'
$ grep '\bpar' anchors.txt
sub par
cart part tart mart

# match words ending with 'par'
$ grep 'par\b' anchors.txt
sub par
spar

# match only whole word 'par'
$ grep '\bpar\b' anchors.txt
sub par
$ grep -w 'par' anchors.txt
sub par
```



Word boundaries behave a bit differently than the `-w` option. See the [Word boundary differences](#) section for details.





Alternatively, you can use `\<` to indicate the start of word anchor and `\>` to indicate the end of word anchor. Using `\b` is preferred as it is more commonly used in other regular expression implementations and has `\B` as its opposite.

## Opposite Word Anchor

The word boundary has an opposite anchor too. `\B` matches wherever `\b` doesn't match. This duality will be seen with some other escape sequences too.

```
# match 'par' if it is surrounded by word characters
$ grep '\Bpar\B' anchors.txt
apparent effort
two spare computers

# match 'par' but not as start of word
$ grep '\Bpar' anchors.txt
spar
apparent effort
two spare computers

# match 'par' but not as end of word
$ grep 'par\B' anchors.txt
apparent effort
two spare computers
cart part tart mart
```



Negative logic is handy in many text processing situations. But use it with care, you might end up matching things you didn't intend.

## Alternation

Many a times, you'd want to search for multiple terms. In a conditional expression, you can use the logical operators to combine multiple conditions. With regular expressions, the `|` metacharacter is similar to logical OR. The regular expression will match if any of the patterns separated by `|` is satisfied.

Alternation is similar to using multiple `-e` option, but provides more flexibility when combined with grouping. The `|` metacharacter syntax varies between BRE and ERE. Quoting from the manual:

In basic regular expressions the meta-characters `?`, `+`, `{`, `|`, `(`, and `)` lose their special meaning; instead use the backslashed versions `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`.

Here are some examples:

```

$ cat pets.txt
I like cats
I like parrots
I like dogs

# three different ways to match either 'cat' or 'dog'
$ grep 'cat\|dog' pets.txt
I like cats
I like dogs
$ grep -E 'cat|dog' pets.txt
I like cats
I like dogs
$ grep -e 'cat' -e 'dog' pets.txt
I like cats
I like dogs

# extract either 'cat' or 'dog' or 'fox' case insensitively
$ printf 'CATs dog bee parrot FoX' | grep -ioE 'cat|dog|fox'
CAT
dog
FoX

```

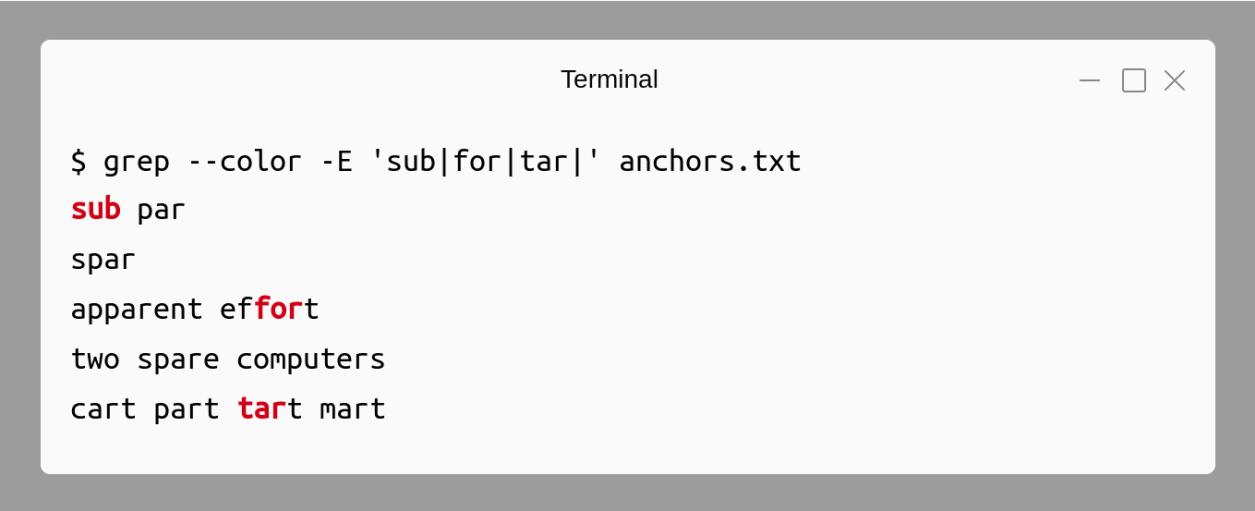
Here's an example of alternate patterns with their own anchors:

```

# match lines starting with 't' or a line containing a word ending with 'ar'
$ grep -E '^t|ar\b' anchors.txt
sub par
spar
two spare computers

```

Sometimes, you want to view the entire input file with only the required search patterns highlighted. You can use an empty alternation to match any line.



A terminal window titled "Terminal" with standard window controls (minimize, maximize, close) in the top right corner. The terminal displays the command `$ grep --color -E 'sub|for|tar|' anchors.txt` and its output. The output lines are: `sub par`, `spar`, `apparent effort`, `two spare computers`, and `cart part tart mart`. In each line, the word matching the search pattern is highlighted in red: `sub`, `effort`, `tart`, and `par` (from `spar`).

```

$ grep --color -E 'sub|for|tar|' anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart

```

## Alternation precedence

There are some tricky corner cases when using alternation. If it is used for filtering a line, there is no ambiguity. However, for matching portion extraction with `-o` option, it depends on a few factors. Say, you want to extract `are` or `spared` — which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else?

The alternative which matches earliest in the input gets precedence.

```
$ echo 'car spared spar' | grep -oE 'are|spared'
spared
$ echo 'car spared spar' | grep -oE 'spared|are'
spared
```

In case of matches starting from same location, for example `party` and `par`, the longest matching portion gets precedence. See [Longest match wins](#) section for more examples. See [regular-expressions: alternation](#) for more information on this topic.

```
# same output irrespective of alternation order
$ echo 'pool party 2' | grep -oE 'party|par'
party
$ echo 'pool party 2' | grep -oE 'par|party'
party

# other implementations like PCRE have left-to-right priority
$ echo 'pool party 2' | grep -oP 'par|party'
par
```

## Grouping

Often, there are some common things among the regular expression alternatives. It could be common characters or qualifiers like the anchors. In such cases, you can group them using a pair of parentheses metacharacters. Similar to  $a(b+c)d = abd+acd$  in maths, you get  $a(b|c)d = abd|acd$  in regular expressions.

```
# without grouping
$ printf 'red\nreform\nread\ncrest' | grep -E 'reform|rest'
reform
crest

# with grouping
$ printf 'red\nreform\nread\ncrest' | grep -E 're(form|st)'
reform
crest

# without grouping
$ grep -E '\bpar\b|\bpart\b' anchors.txt
sub par
cart part tart mart

# taking out common anchors
$ grep -E '\b(par|part)\b' anchors.txt
sub par
cart part tart mart
```

```
# taking out common characters as well
# you'll later learn a better technique instead of using empty alternate
$ grep -E '\bpar(|t)\b' anchors.txt
sub par
cart part tart mart
```

## Escaping metacharacters

You have already seen a few metacharacters and escape sequences that help compose a regular expression. To match the metacharacters literally, i.e. to remove their special meaning, prefix those characters with a `\` character. To indicate a literal `\` character, use `\\`. Some of the metacharacters, like the line anchors, lose their special meaning when not used in their customary positions with BRE syntax.

If there are many metacharacters to be escaped, try to work out alternate solutions by using `-F` (paired with regular expression like options such as `-e`, `-f`, `-i`, `-w`, `-x`, etc) or by switching between ERE and BRE. Another option is to use PCRE (covered later), which has special constructs to mark whole or portion of pattern to be matched literally — especially useful when using shell variables.

```
# line anchors aren't special away from customary positions with BRE
$ echo 'a^2 + b^2 - C*3' | grep 'b^2'
a^2 + b^2 - C*3
$ echo '$a = $b + $c' | grep '$b'
$a = $b + $c

# escape line anchors to match literally if you are using ERE
# or if you want to match them at customary positions with BRE
$ echo '$a = $b + $c' | grep -o '\$' | wc -l
3

# or use -F where possible
$ echo '$a = $b + $c' | grep -oF '$' | wc -l
3
```

Here's another example to show differences between BRE and ERE:

```
# cannot use -F here as line anchor is needed
$ printf '(a/b) + c\n3 + (a/b) - c' | grep '^ (a/b)'
(a/b) + c
$ printf '(a/b) + c\n3 + (a/b) - c' | grep -E '^ (a/b)'
(a/b) + c
```

## Matching characters like tabs

GNU `grep` doesn't support escape sequences like `\t` (tab) and `\n` (newline). Neither does it support formats like `\xNN` (specifying a character by its codepoint value in hexadecimal format). Shells like Bash support [ANSI-C Quoting](#) as an alternate way to use such escape sequences.

```
# '$' is ANSI-C quoting syntax
$ printf 'go\tto\ngo to' | grep $'go\tto'
go      to

# \x20 in hexadecimal represents the space character
$ printf 'go\tto\ngo to' | grep $'go\x20to'
go to
```



Undefined escape sequences are treated as the character it escapes. Newer versions of `GNU grep` will generate a warning for such escapes and might become errors in future versions.

```
$ echo 'sea eat car rat eel tea' | grep 's\ea'
grep: warning: stray \ before e
sea eat car rat eel tea
```

## The dot metacharacter

The dot metacharacter serves as a placeholder to match any character. Later you'll learn how to define your own custom placeholders for a limited set of characters.

```
# extract 'c', followed by any character and then 't'
$ echo 'tac tin cot abc:tuv excite' | grep -o 'c.t'
c t
cot
c:t
cit

$ printf '42\t33\n'
42      33

# extract '2', followed by any character and then '3'
$ printf '42\t33\n' | grep -o '2.3'
2      3
```

If you are using a Unix-like distribution, you'll likely have the `/usr/share/dict/words` dictionary file. This will be used as an input file to illustrate regular expression examples in this chapter. This file is included in the [learn\\_gnugrep\\_ripgrep repo](#) as `words.txt` file (modified to make it ASCII only).

```
$ wc -l words.txt
98927 words.txt

# 5 character lines starting with 'du' and ending with 'ts' or 'ky'
$ grep -xE 'du.(ky|ts)' words.txt
ducts
duets
dusky
dusts
```

## Quantifiers

Alternation helps you match one among multiple patterns. Combining the dot metacharacter with quantifiers (and alternation if needed) paves a way to perform logical AND between patterns. For example, to check if a string matches two patterns with any number of characters in between. Quantifiers can be applied to characters, groupings and some more constructs that'll be discussed later. Apart from the ability to specify exact quantity and bounded range, these can also match unbounded varying quantities.

BRE/ERE support only one type of quantifiers, whereas PCRE supports three types. Quantifiers in `GNU grep` behave mostly like greedy quantifiers supported by PCRE, but there are subtle differences, which will be discussed with examples later on.

First up, the `?` metacharacter which quantifies a character or group to match `0` or `1` times. This helps to define optional patterns and build terser patterns compared to alternation and groupings for some cases.

```
# same as: grep -E '\b(fe.d|fed)\b'
# BRE version: grep -w 'fe.\?d'
$ printf 'fed\nfod\nfe:d\nfeed' | grep -wE 'fe.?d'
fed
fe:d
feed

# same as: grep -E '\bpar(|t)\b'
$ printf 'sub par\nspare\npart time' | grep -wE 'part?'
sub par
part time

# same as: grep -oE 'part|parrot'
$ echo 'par part parrot parent' | grep -oE 'par(ro)?t'
part
parrot
# same as: grep -oE 'part|parrot|parent'
$ echo 'par part parrot parent' | grep -oE 'par(en|ro)?t'
part
parrot
parent
```

The `*` metacharacter quantifies a character or group to match `0` or more times.

```
# extract 'f' followed by zero or more of 'e' followed by 'd'
$ echo 'fd fed fod fe:d feeeder' | grep -o 'fe*d'
fd
fed
feeeder

# extract zero or more of '1' followed by '2'
$ echo '311111111125111142' | grep -o '1*2'
1111111112
2
```

The `+` metacharacter quantifies a character or group to match `1` or more times.

```
# extract 'f' followed by one or more of 'e' followed by 'd'
# BRE version: grep -o 'fe\+d'
$ echo 'fd fed fod fe:d feeeeder' | grep -oE 'fe+d'
fed
feeed

# extract 'f' followed by at least one of 'e' or 'o' or ':' followed by 'd'
$ echo 'fd fed fod fe:d feeeeder' | grep -oE 'f(e|o|:)+d'
fed
fod
fe:d
feeed

# extract one or more of '1' followed by '2'
$ echo '311111111125111142' | grep -oE '1+2'
1111111112

# extract one or more of '1' followed by optional '4' and then '2'
$ echo '311111111125111142' | grep -oE '1+4?2'
1111111112
111142
```

You can specify a range of integer numbers, both bounded and unbounded, using `{}` metacharacters. There are four ways to use this quantifier as listed below:

Quantifier	Description
<code>{m,n}</code>	match <code>m</code> to <code>n</code> times
<code>{m,}</code>	match at least <code>m</code> times
<code>{,n}</code>	match up to <code>n</code> times (including <code>0</code> times)
<code>{n}</code>	match exactly <code>n</code> times

```
# note that stray characters like space is not allowed anywhere within {}
# BRE version: grep -o 'ab\{1,4\}c'
$ echo 'abc ac adc abbc xabbbcz bbb bc abbbbbc' | grep -oE 'ab{1,4}c'
abc
abbc
abbbc

$ echo 'abc ac adc abbc xabbbcz bbb bc abbbbbc' | grep -oE 'ab{3,}c'
abbbc
abbbbbc

$ echo 'abc ac adc abbc xabbbcz bbb bc abbbbbc' | grep -oE 'ab{,2}c'
abc
ac
abbc

$ echo 'abc ac adc abbc xabbbcz bbb bc abbbbbc' | grep -oE 'ab{3}c'
abbbc
```



To match `{}` metacharacters literally (assuming ERE), escaping `{` alone is enough. Or if it doesn't conform strictly to any of the four forms listed above, escaping is not needed at all.

```
$ echo 'a{5} = 10' | grep -E 'a\{5}'
a{5} = 10

$ echo 'report_{a,b}.txt' | grep -E '_{a,b}'
report_{a,b}.txt
```

## Conditional AND

Next up, constructing AND conditional using dot metacharacter and quantifiers. To allow matching in any order, you'll have to bring in alternation as well. That is somewhat manageable for 2 or 3 patterns. With PCRE, you can use [lookarounds for a comparatively easier approach](#).

```
# match 'Error' followed by zero or more characters followed by 'valid'
$ echo 'Error: not a valid input' | grep -o 'Error.*valid'
Error: not a valid

$ echo 'cat and dog and parrot' | grep -oE 'cat.*dog|dog.*cat'
cat and dog
$ echo 'dog and cat and parrot' | grep -oE 'cat.*dog|dog.*cat'
dog and cat
```

## Longest match wins

You've already seen an example where the longest matching portion was chosen if the alternatives started from the same location. For example `spar|spared` will result in `spared` being chosen over `spar`. The same applies whenever there are two or more matching possibilities from same starting location. For example, `f.?o` will match `foo` instead of `fo` if the input string to match is `foot`.

```
# longest match among 'foo' and 'fo' wins here
$ echo 'foot' | grep -oE 'f.?o'
foo

# everything will match here
$ echo 'car bat cod map scat dot abacus' | grep -o '.*'
car bat cod map scat dot abacus

# longest match happens when (1|2|3)+ matches up to '1233' only
# so that '12apple' can match as well
$ echo 'fig123312apple' | grep -oE 'g(1|2|3)+(12apple)?'
g123312apple

# in other implementations like PCRE, that is not the case
# precedence is left to right for greedy quantifiers
$ echo 'fig123312apple' | grep -oP 'g(1|2|3)+(12apple)?'
g123312
```



While determining the longest match, the overall regular expression matching is also considered. That's how `Error.*valid` example worked. If `.*` had consumed everything after `Error`, there wouldn't be any more characters to try to match `valid`. So, among the varying quantity of characters to match for `.*`, the longest portion that satisfies the overall regular expression is chosen. Something like `a.*b` will match from the first `a` in the input string to the last `b`. In other implementations, like PCRE, this is achieved through a process called **backtracking**. These approaches have their own advantages and disadvantages and have cases where the pattern can result in exponential time consumption.

```
# extract from the start of the line to the last 'm' in the line
$ echo 'car bat cod map scat dot abacus' | grep -o '.*m'
car bat cod m

# extract from the first 'c' to the last 't' in the line
$ echo 'car bat cod map scat dot abacus' | grep -o 'c.*t'
car bat cod map scat dot

# extract from the first 'c' to the last 'at' in the line
$ echo 'car bat cod map scat dot abacus' | grep -o 'c.*at'
car bat cod map scat

# here 'm*' will match 'm' zero times as that gives the longest match
$ echo 'car bat cod map scat dot abacus' | grep -o 'b.*m*'
bat cod map scat dot abacus
```

## Character classes

To create a custom placeholder for limited set of characters, enclose them inside `[]` metacharacters. It is similar to using single character alternations inside a grouping, but with added flexibility and features. Character classes have their own versions of metacharacters and provide special predefined sets for common use cases. Quantifiers are also applicable to character classes.

```
# same as: grep -E 'cot|cut' or grep -E 'c(o|u)t'
$ printf 'cute\ncat\ncot\nccoat\ncost\nscuttle' | grep 'c[ou]t'
cute
cot
scuttle

# same as: grep -E '(a|e|o)+t'
$ printf 'meeting\ncute\nboat\nsite\nfoot' | grep -E '[aeo]+t'
meeting
boat
foot

# same as: grep -owE '(s|o|t)(o|n)'
$ echo 'do so in to no on' | grep -ow '[sot][on]'
so
to
on
```

```
# lines made up of letters 'o' and 'n', line length at least 2
$ grep -xE '[on]{2,}' words.txt
no
non
noon
on
```

## Character class metacharacters

Character classes have their own metacharacters to help define the sets succinctly. Metacharacters outside of character classes like `^`, `$`, `()` etc either don't have special meaning or have a completely different one inside the character classes.

First up, the `-` metacharacter that helps to define a range of characters instead of having to specify them all individually.

```
# same as: grep -oE '[0123456789]+'
$ echo 'Sample123string42with777numbers' | grep -oE '[0-9]+'
123
42
777

# whole words made up of lowercase alphabets only
$ echo 'coat Bin food tar12 best' | grep -owE '[a-z]+'
coat
food
best

# whole words made up of lowercase alphabets and digits only
$ echo 'coat Bin food tar12 best' | grep -owE '[a-z0-9]+'
coat
food
tar12
best

# whole words made up of lowercase alphabets, starting with 'p' to 'z'
$ echo 'go no u grip read eat pit' | grep -owE '[p-z][a-z]*'
u
read
pit
```

Character classes can also be used to construct numeric ranges. However, it is easy to miss corner cases and some ranges are complicated to construct.

```
# numbers between 10 to 29
$ echo '23 154 12 26 34' | grep -ow '[12][0-9]'
23
12
26
```

```
# numbers >= 100
$ echo '23 154 12 26 98234' | grep -owE '[0-9]{3,}'
154
98234

# numbers >= 100 if there are leading zeros
$ echo '0501 035 154 12 26 98234' | grep -owE '0*[1-9][0-9]{2,}'
0501
154
98234
```

Next metacharacter is `^` which has to be specified as the first character of the character class. It negates the set of characters, so all characters other than those specified will be matched. As highlighted earlier, handle negative logic with care, you might end up matching more than you wanted.

```
# all non-digits
$ echo 'Sample123string42with777numbers' | grep -oE '[^0-9]+'
Sample
string
with
numbers

# extract characters from the start of string based on a delimiter
$ echo 'apple:123:banana:cherry' | grep -o '[^:]*'
apple

# extract last two columns based on a delimiter
$ echo 'apple:123:banana:cherry' | grep -oE '(:[^:]+){2}$'
:banana:cherry

# get all sequence of characters surrounded by double quotes
$ echo 'I like "mango" and "guava"' | grep -oE '"[^"]+"'
"mango"
"guava"
```

Sometimes, it is easier to use positive character class and the `-v` option instead of using negated character classes.

```
# lines not containing vowel characters
# note that this will match empty lines too
$ printf 'tryst\nfun\nglyph\npity\nwhy' | grep -xE '[^aeiou]*'
tryst
glyph
why

# easier to write and maintain
$ printf 'tryst\nfun\nglyph\npity\nwhy' | grep -v '[aeiou]'
tryst
glyph
why
```

## Escape sequence sets

Some commonly used character sets have predefined escape sequences:

- `\w` matches all **word** characters `[a-zA-Z0-9_]` (recall `-w` definition)
- `\W` matches all non-word characters (recall duality seen earlier, like `\b` and `\B`)
- `\s` matches all **whitespace** characters: tab, newline, vertical tab, form feed, carriage return and space
- `\S` matches all non-whitespace characters

These escape sequences cannot be used inside character classes (unlike PCRE). Also, as mentioned earlier, these definitions assume ASCII input.

```
# extract all word character sequences
$ printf 'load;err_msg--\nant,r2..not\n' | grep -o '\w*'
load
err_msg
ant
r2
not

$ echo 'sea eat car rat eel tea' | grep -o '\b\w' | paste -sd ''
secret

# extract all non-whitespace character sequences
$ printf '  1..3 \v\f fig_tea 42\tzzz \r\n1-2-3\n\n' | grep -o '\S*'
1..3
fig_tea
42
zzz
1-2-3
```

## Named character sets

A **named character set** is defined by a name enclosed between `[:` and `:]` and has to be used within a character class `[]`, along with other characters as needed.

Named set	Description
<code>[:digit:]</code>	<code>[0-9]</code>
<code>[:lower:]</code>	<code>[a-z]</code>
<code>[:upper:]</code>	<code>[A-Z]</code>
<code>[:alpha:]</code>	<code>[a-zA-Z]</code>
<code>[:alnum:]</code>	<code>[0-9a-zA-Z]</code>
<code>[:xdigit:]</code>	<code>[0-9a-fA-F]</code>
<code>[:cntrl:]</code>	control characters — first 32 ASCII characters and 127th (DEL)
<code>[:punct:]</code>	all the punctuation characters
<code>[:graph:]</code>	<code>[:alnum:]</code> and <code>[:punct:]</code>
<code>[:print:]</code>	<code>[:alnum:]</code> , <code>[:punct:]</code> and space
<code>[:blank:]</code>	space and tab characters
<code>[:space:]</code>	whitespace characters, same as <code>\s</code>

Here are some examples:

```
$ printf 'err_msg\xerox\nant\nm_2\nP2\nload1\neel' | grep -x '[:lower:]*'
xerox
ant
eel

$ printf 'err_msg\xerox\nant\nm_2\nP2\nload1\neel' | grep -x '[:lower:]*_*'
err_msg
xerox
ant
eel

$ printf 'err_msg\xerox\nant\nm_2\nP2\nload1\neel' | grep -x '[:alnum:]*'
xerox
ant
P2
load1
eel

$ echo 'pie tie#ink-eat_42;' | grep -o '^[[:punct:]]*'
pie tie
ink
eat
42
```

## Matching character class metacharacters literally

Specific placement is needed to match the character class metacharacters literally.

- should be the first or the last character.

```
# same as: grep -owE '[-a-z]{2,}'
$ echo 'ab-cd gh-c 12-423' | grep -owE '[a-z-]{2,}'
ab-cd
gh-c
```

- ] should be the first character.

```
# no match
$ printf 'int a[5]\nfig\n1+1=2\n' | grep '[=]']

# correct usage
$ printf 'int a[5]\nfig\n1+1=2\n' | grep '[[]=]'
int a[5]
1+1=2
```

- [ can be used anywhere in the character set, but not combinations like [. or [:. Using [][] will match both [ and ] .

```
$ echo 'int a[5]' | grep '[x[.y]']
grep: Unmatched [, [^, [:, [., or [=
```

```
$ echo 'int a[5]' | grep '[x[y.]'
int a[5]
```

`^` should be other than the first character.

```
$ echo 'f*(a^b) - 3*(a+b)/(a-b)' | grep -o 'a[+^]b'
a^b
a+b
```

Characters like `\` and `$` are not special.

```
$ echo '5ba\babc2' | grep -o '[a\b]*'
ba\bab
```



As seen in the examples above, combinations like `[.` or `[:` cannot be used together to mean two individual characters, as they have special meaning within `[]`. See [Character Classes and Bracket Expressions](#) section in `info grep` for more details.

## Backreferences

The grouping metacharacters `()` are also known as **capture groups**. Similar to variables in programming languages, the portion captured by `()` can be referred later using backreferences. The syntax is `\N` where `N` is the capture group you want. Leftmost `(` in the regular expression is `\1`, next one is `\2` and so on up to `\9`.

```
# 8 character lines having same 3 lowercase letters at the start and end
$ grep -xE '([a-z]{3})..\\1' words.txt
mesdames
respires
restores
testates
# different than: grep -xE '([a-d]..){2}'
$ grep -xE '([a-d]..)\\1' words.txt
bonbon
cancan
chichi

# whole words that have at least one consecutive repeated character
$ echo 'effort flee facade oddball rat tool' | grep -owE '\\w*(\\w)\\1\\w*'
effort
flee
oddball
tool

# spot repeated words
# use \\s instead of \\W if only whitespaces are allowed between words
$ printf 'spot the the error\\nno issues here' | grep -wE '\\(\\w+)\\W+\\1'
spot the the error
```



Backreference will provide the string that was matched, not the pattern that was inside the capture group. For example, if `([0-9][a-f])` matches `3b`, then backreferencing will give `3b` and not any other valid match like `8f`, `0a` etc. This is akin to how variables behave in programming, only the result of expression stays after variable assignment, not the expression itself.

## Known Bugs

Visit [grep bug list](#) for a list of known issues. See [GNU grep manual: Known Bugs](#) for a list of backreference related bugs.

Large repetition counts in the `{n,m}` construct may cause grep to use lots of memory. In addition, certain other obscure regular expressions require exponential time and space, and may cause grep to run out of memory.

Back-references can greatly slow down matching, as they can generate exponentially many matching possibilities that can consume both time and memory to explore. Also, the POSIX specification for back-references is at times unclear. Furthermore, many regular expression implementations have back-reference bugs that can cause programs to return incorrect answers or even crash, and fixing these bugs has often been low-priority

Here's an [issue for certain usage of backreferences and quantifier](#) that was filed by yours truly.

```
# takes some time and results in no output
# aim is to get words having two occurrences of repeated characters
$ grep -m5 -xiE '([a-z]*([a-z])\2[a-z]*){2}' words.txt
# works when the nesting is unrolled
$ grep -m5 -xiE '[a-z]*([a-z])\1[a-z]*([a-z])\2[a-z]*' words.txt
Abbott
Annabelle
Annette
Appaloosa
Appleseed

# no problem if PCRE is used
$ grep -m5 -xiP '([a-z]*([a-z])\2[a-z]*){2}' words.txt
Abbott
Annabelle
Annette
Appaloosa
Appleseed
```

[unix.stackexchange: Why doesn't this sed command replace the 3rd-to-last "and"?](#) shows another interesting bug when word boundaries and group repetitions are involved. Some examples are shown below. Again, workaround is to use PCRE or expand the group.

```
# wrong output
$ echo 'cocoa' | grep -E '(\bco){2}'
```

```
cocoa
# correct behavior, no output
$ echo 'cocoa' | grep -E '\bco\bco'
$ echo 'cocoa' | grep -P '(\bco){2}'

# wrong output
$ echo 'it line with it here sit too' | grep -oE 'with(.*\bit\b){2}'
with it here sit
# correct behavior, no output
$ echo 'it line with it here sit too' | grep -oE 'with.*\bit\b.*\bit\b'
$ echo 'it line with it here sit too' | grep -oP 'with(.*\bit\b){2}'
```

Changing word boundaries to `\<` and `\>` results in a different issue:

```
# this correctly gives no output
$ echo 'it line with it here sit too' | grep -oE 'with(.*\<it\>){2}'
# this correctly gives output
$ echo 'it line with it here it too' | grep -oE 'with(.*\<it\>){2}'
with it here it

# but this one fails
$ echo 'it line with it here it too sit' | grep -oE 'with(.*\<it\>){2}'
# correct behavior
$ echo 'it line with it here it too sit' | grep -oP 'with(.*\bit\b){2}'
with it here it
```

## Summary

Mastering regular expressions is not only important for using `grep` effectively, but also comes in handy for text processing with other CLI tools like `sed` and `awk` and programming languages like `Python` and `Ruby`. These days, some of the GUI applications also support regular expressions. One main thing to remember is that syntax and features will vary. This book itself discusses four variations — BRE, ERE, PCRE and `ripgrep` regex. However, core concepts are likely to be same and having a handy reference sheet would go a long way in reducing misuse.

## Exercises



The [exercises](#) directory has all the files used in this section.

**1)** For the input file `patterns.txt`, extract from `(` to the next occurrence of `)` unless they contain parentheses characters in between.

```
##### add your solution here
(division)
(#modulo)
(9-2)
()
(j/k-3)
```



```
(greeting)
(b)
```

2) For the input file `patterns.txt` , match all lines that start with `den` or end with `ly` .

```
##### add your solution here
2 lonely
dent
lovely
```

3) For the input file `patterns.txt` , extract all whole words containing `42` surrounded by word characters on both sides.

```
##### add your solution here
Hi42Bye
nice1423
cool_42a
_42_
```

4) For the input file `patterns.txt` , match all lines containing `car` but not as a whole word.

```
##### add your solution here
scar
care
a huge discarded pile of books
scare
part cart mart
```

5) Count the total number of times the whole words `removed` or `rested` or `received` or `replied` or `refused` or `retired` are present in the `patterns.txt` file.

```
##### add your solution here
9
```

6) For the input file `patterns.txt` , match lines starting with `s` and containing `e` and `t` in any order.

```
##### add your solution here
sets tests
site cite kite bite
subtle sequoia
```

7) From the input file `patterns.txt` , extract all whole lines having the same first and last word character.

```
##### add your solution here
sets tests
Not a pip DOWN
y
1 dentist 1
_42_
```

8) For the input file `patterns.txt` , match all lines containing `*[5]` literally.

```
##### add your solution here
(9-2)*[5]
```

**9)** For the given quantifiers, what would be the equivalent form using the `{m,n}` representation?

- `?` is same as
- `*` is same as
- `+` is same as

**10)** In ERE, `(a*|b*)` is same as `(a|b)*` — True or False?

**11)** `grep -wE '[a-z](on|no)[a-z]'` is same as `grep -wE '[a-z][on]{2}[a-z]'` . True or False? Sample input shown below might help to understand the differences, if any.

```
$ printf 'known\nmood\nknow\npony\ninns\n'
known
mood
know
pony
inns
```

**12)** For the input file `patterns.txt` , display all lines starting with `hand` and ending immediately with `s` or `y` or `le` or no further characters.

```
##### add your solution here
handle
handy
hands
hand
```

**13)** For the input files `patterns.txt` , display matching lines based on the patterns (one per line) present in the `regex_terms.txt` file.

```
$ cat regex_terms.txt
^[c-k].*\W$
ly.
[A-Z].*[0-9]
```

```
##### add your solution here
Hi42Bye nice1423 bad42
fly away
def factorial()
hand
```

**14)** Will the ERE pattern `^a\w+([0-9]+:fig)?` match the same characters for the input `apple42:banana314` and `apple42:fig100` ? If not, why not?

**15)** For the input file `patterns.txt` , match all lines starting with `[5]` .

```
##### add your solution here
[5]*3
```

**16)** What characters will the pattern `\t` match? A tab character or `\` followed by a `t` or something else? Does the behavior change inside a character class? What alternatives are

there to match a tab character?

**17)** From the input file `patterns.txt` , extract all hexadecimal sequences with a minimum of four characters. Match `0x` as an optional prefix, but shouldn't be counted for determining the length. Match the characters case insensitively, and the sequences shouldn't be surrounded by other word characters.

```
##### add your solution here
0XdeadBEEF
bad42
0x0ff1ce
```

**18)** From the input file `patterns.txt` , extract from `-` till the end of the line, provided the characters after the hyphen are all word characters only.

```
##### add your solution here
-handy
-icy
```

**19)** For the input file `patterns.txt` , count the total number of lines containing `e` or `i` followed by `l` or `n` and vice versa.

```
##### add your solution here
18
```

**20)** For the input file `patterns.txt` , match lines starting with `4` or `-` or `u` or `sub` or `care` .

```
##### add your solution here
care
4*5]
-handy
subtle sequoia
unhand
```