learnbyexample

# GNU GREP and RIPGREP

## *The search is on*

*Sundeep Agarwal*

# Table of contents

# Preface

You are likely to be familiar with using `Ctrl+F` from an editor, word processor, web browser, IDE, etc to quickly locate where a particular string occurs. `grep` is similar, but much more versatile and feature-rich version of the search functionality usable from command line. Modern requirements have given rise to tools like `ripgrep` that provide out-of-box features such as recursive search while respecting ignore rules of a version controlled directory. An important feature that the GUI applications may lack is **regular expressions**, which helps to precisely define a matching criteria. You could consider it as sort of a mini-programming language in itself. So, apart from covering command options, regular expressions will also be discussed in detail.

The book heavily leans on examples to present features one by one. It is recommended that you manually type each example and experiment with them. Understanding both the nature of sample input string and the output produced is essential. As an analogy, consider learning to drive a bike or a car — no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

My Command Line Text Processing repository includes a chapter on `GNU grep` which has been edited and expanded to create this book.

## Prerequisites

Prior experience working with command line and `bash` shell, should know concepts like file redirection, command pipeline and so on.

If you are new to the world of command line, check out ryanstutorials or my GitHub repository on Linux Command Line before starting this book.

## Conventions

- The examples presented here have been tested on `GNU bash` shell with **GNU grep 3.4** and **ripgrep 12.1.0** 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, output is skipped for commands like `wget` and so on
- Unless otherwise noted, all examples and explanations are meant for **ASCII** characters
- 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_gnugrep_ripgrep repo has all the files used in examples and exercises and other details related to the book. If you are not familiar with `git` command, click the **Clone or download** 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 `bash` , `grep` and other commands
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- Cover image
    - LibreOffice Draw
    - detective by olarte.ollie under Creative Commons Attribution-Share Alike 2.0 Generic
- softwareengineering.stackexchange and skolakoda for programming quotes
- Warning and Info icons by Amada44 under public domain
- Andrew Gallant (author of `ripgrep` ) and mikeblas 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 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_gnugrep_ripgrep/issues

Goodreads: https://www.goodreads.com/book/show/47406700-gnu-grep-and-ripgrep

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.

# 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**lobally search a **r**egular **e**xpression and **p**rint), which has the same effect: doing a global search with the regular expression and printing all matching lines. Grep was originally developed for the Unix operating system, but later available for all Unix-like systems and some others such as OS-9.

Use of `grep` has become so ubiquitous that it has found its way into Oxford dictionary as well. As part of daily computer usage, searching comes up often. It could be finding the right emoji by their name on social media or using `Ctrl+F` in word processor/IDE/web browser/ebook/bookmarks/etc. Some of these have options to refine the search further like matching case, ignoring case, limiting to whole word, using regular expressions etc. `grep` provides all of these features and much more when it comes to searching or extracting content from text files. After getting used to `grep`, the search features provided by GUI programs often feel inadequate and slower.

## Installation

If you are on a Unix like system, you are most likely to already have some version of `grep` installed. This book is primarily for `GNU grep` and also has a chapter on `ripgrep`. As there are syntax and feature differences between various implementations, please make sure to follow along with what is presented here. `GNU grep` is part of text creation and manipulation commands provided by `GNU` and comes by default on GNU/Linux. To install newer or particular version, visit gnu: software and check release notes for an overview of changes between versions. See also bug list.

```
$ wget https://ftp.gnu.org/gnu/grep/grep-3.4.tar.xz
$ tar -Jxf grep-3.4.tar.xz
$ cd grep-3.4/
$ ./configure
$ make
$ sudo make install

$ type -a grep
grep is /usr/local/bin/grep
grep is /bin/grep
$ grep -V | head -n1
grep (GNU grep) 3.4
```

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

- git-bash
- WSL
- brew

## Options overview

It is always a good idea to know where to find the documentation. From command line, you can use `man grep` for a short manual and `info grep` for full documentation. For a better interface, visit online gnu grep manual.

```
$ man grep
NAME
       grep, egrep, fgrep - 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 patterns
       separated by newline characters, and grep prints each line that matches
       a pattern.

       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:

**Regexp selection**:

| Option | Description |
|---|---|
| -E, --extended-regexp | PATTERN is an extended regular expression (ERE) |
| -F, --fixed-strings | PATTERN is a set of newline-separated strings |
| -G, --basic-regexp | PATTERN is a basic regular expression (BRE) |
| -P, --perl-regexp | PATTERN is a Perl regular expression |
| -e, --regexp=PATTERN | use PATTERN for matching |
| -f, --file=FILE | obtain PATTERN from FILE |
| -i, --ignore-case | ignore case distinctions |
| -w, --word-regexp | force PATTERN to match only whole words |
| -x, --line-regexp | force PATTERN to 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 matches |
| -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 the file name for each match |
| -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 the part of a line matching PATTERN |
| -q, --quiet, --silent | suppress all normal output |
| --binary-files=TYPE | assume that binary files are TYPE;<br>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;<br>ACTION is ’read’, ’recurse’, or ’skip’ |
| -D, --devices=ACTION | how to handle devices, FIFOs and sockets;<br>ACTION is ’read’ or ’skip’ |
| -r, --recursive | like --directories=recurse |
| -R, --dereference-recursive | likewise, but follow all symlinks |
| --include=FILE_PATTERN | search only files that match FILE_PATTERN |
| --exclude=FILE_PATTERN | skip files and directories matching FILE_PATTERN |
| --exclude-from=FILE | skip files matching any file pattern from FILE |
| --exclude-dir=PATTERN | directories that match PATTERN will be skipped. |
| -L, --files-without-match | print only names of FILEs containing no match |
| -l, --files-with-matches | print only names of FILEs containing matches |
| -c, --count | print only a count of matching 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 |
| --color[=WHEN],<br>--colour[=WHEN] | use markers to highlight the matching strings;<br>WHEN is ’always’, ’never’, or ’auto’ |
| -U, --binary | do not strip CR characters at EOL (MSDOS/Windows) |
| -u, --unix-byte-offsets | report offsets as if CRs were not there<br>(MSDOS/Windows) |

# Frequently used options

This chapter will cover many of the options provided by `GNU grep` . Regular expressions will be covered from next chapter, so the examples in this chapter will use only literal strings for input patterns. Literal or fixed string matching means exact string comparison is intended, no special meaning for any character.


Files used in examples are available chapter wise from learn_gnugrep_ripgrep repo. The directory for this chapter is `freq_options` .

## Simple string search

By default, `grep` would print all input *lines* which matches the given search patterns. The newline character `\n` is considered as the line separator. This section will show you how to filter lines matching a given search string using `grep` .

```
$ # sample input file for this section
$ 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
```

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

```
$ grep 'twice' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.

$ grep 'e th' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
A language that does not affect the way you think about programming,
```

If the filename is `-` or left out, `grep` will perform the search on `stdin` data.

```
$ printf 'avocado\nmango\nguava' | grep 'v'
avocado
guava
```

 If your input file has some other format like `\r\n` (carriage return and newline characters) as 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. Make sure to remember this point, it'll come up in exercises.

```
$ # Unix and DOS style line endings
$ printf '42\n' | file -
/dev/stdin: ASCII text
$ 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 next chapter. For now, use the `-F` option to indicate that the patterns should be matched literally. As a performance optimization, `GNU grep` automatically tries to perform literal search even if `-F` option is not used depending upon the kind of search string specified.

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

## Case insensitive search

Sometimes, you don't know if the log file has `error` or `Error` or `ERROR` and so on. In such cases, you can use `-i` option to ignore case.

```
$ grep -i 'jam' programming_quotes.txt
use regular expressions. Now they have two problems by Jamie Zawinski

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

## Invert matching lines

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

```
$ seq 5 | grep -v '3'
1
2
4
5

$ printf 'goal\nrate\neat\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 expression, etc in coming sections.

## Line number and count

The `-n` option will prefix line number and a colon character while displaying the output results. This is useful to quickly locate the matching lines for further editing.

```
$ grep -n 'not' programming_quotes.txt
3:by definition, not smart enough to debug it by Brian W. Kernighan
8:A language that does not affect the way you think about programming,
9:is not worth knowing by Alan Perlis

$ printf 'great\nnumber\numpteen' | grep -n 'r'
1:great
2:number
```

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

```
$ # count of lines matching the pattern
$ grep -c 'in' programming_quotes.txt
8

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

With multiple file input, count is displayed for each file separately. Use `cat` if you need a combined count.

```
$ # here - is placeholder for stdin data
$ seq 15 | grep -c '1' programming_quotes.txt -
programming_quotes.txt:1
(standard input):7
```

```
$ # useful application of cat command
$ cat <(seq 15) programming_quotes.txt | grep -c '1'
8
```


The output given by `-c` is total number of **lines** matching the given patterns, not total number of matches. Use `-o` option and pipe the output to `wc -l` to get total matches (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` would stop processing an input file as soon as condition specified by `-m` is satisfied. Just like `-c` option, *note* that `-m` works by line count, not by number of matches.

```
$ grep -m3 'in' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
by definition, not smart enough to debug it by Brian W. Kernighan
Some people, when confronted with a problem, think - I know, I will

$ 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 '1' or 'two'
$ grep -e '1' -e 'two' programming_quotes.txt
use regular expressions. Now they have two problems by Jamie Zawinski
naming things, and off-by-1 errors by Leon Bambrick
```

If there are lot of search strings, save them in a file (one search string per line and make sure there are no empty lines). Use `-f` option to specify a file as source of search strings. You can use this option multiple times and also add more patterns from command line using the `-e` option.

```
$ printf 'two\n1\n' > search_strings.txt
$ cat search_strings.txt
two
1
```

```
$ grep -f search_strings.txt programming_quotes.txt
use regular expressions. Now they have two problems by Jamie Zawinski
naming things, and off-by-1 errors by Leon Bambrick

$ grep -f search_strings.txt -e 'twice' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
use regular expressions. Now they have two problems by Jamie Zawinski
naming things, and off-by-1 errors by Leon Bambrick
```

To find lines matching all of the search strings, you'd need to resort to regular expressions (covered later) or workaround by using shell pipes. This is similar to **conditional AND** boolean logic.

```
$ # match lines containing both 'in' and 'not' in any order
$ # same as: grep 'not' programming_quotes.txt | grep 'in'
$ grep 'in' programming_quotes.txt | grep 'not'
by definition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
```

## 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, passed to another command like `sed/awk/perl/sort/etc` for further processing and so on. Some of these commands can handle search by themselves, but `grep` is fast and specialized tool for searching and using shell pipes can improve performance if parallel processing is available. Similar to `-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

```
$ # list filename if it contains 'are' anywhere in the file
$ grep -l 'are' programming_quotes.txt search_strings.txt
programming_quotes.txt
$ # no output because no match was found
$ grep -l 'xyz' programming_quotes.txt search_strings.txt
$ # list filename if it contains '1' anywhere in the file
$ grep -l '1' programming_quotes.txt search_strings.txt
programming_quotes.txt
search_strings.txt

$ # list filename if it does NOT contain 'xyz' anywhere in the file
$ grep -L 'xyz' programming_quotes.txt search_strings.txt
programming_quotes.txt
search_strings.txt
$ grep -L 'are' programming_quotes.txt search_strings.txt
search_strings.txt
```

## Filename prefix for matching lines

If there are multiple file inputs, `grep` would automatically prefix filename while displaying matching lines. You can also control whether or not to add the prefix using options.

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

```
$ # -h is on by default for single file input
$ grep '1' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick
$ # using -h to suppress filename prefix
$ seq 1000 | grep -h -m3 '1' - programming_quotes.txt
1
10
11
naming things, and off-by-1 errors by Leon Bambrick

$ # -H is on by default for multiple file input
$ seq 1000 | grep -m3 '1' - programming_quotes.txt
(standard input):1
(standard input):10
(standard input):11
programming_quotes.txt:naming things, and off-by-1 errors by Leon Bambrick
$ # using -H to always show filename prefix
$ grep -H '1' programming_quotes.txt
programming_quotes.txt:naming things, and off-by-1 errors by Leon Bambrick
```

The `vim` editor has an option `-q` that allows to easily edit the matching lines from `grep` output if it has both line number and filename prefixes. Let me know if your favorite editor has equivalent feature, I'll update it here.

```
$ grep -Hn '1' *
programming_quotes.txt:12:naming things, and off-by-1 errors by Leon Bambrick
search_strings.txt:2:1

$ # use :cn and :cp to navigate to next/previous occurrences
$ # the status line at bottom will have additional info
$ # use -H option to ensure filename is always present in output
$ vim -q <(grep -Hn '1' *)
```

## Colored output

When working from terminal, having `--color` option enabled makes it easier to spot the matching portions in output. Especially when experimenting to get the correct regular expression. Modern terminal will usually have color support, see unix.stackexchange: How to check if bash can print colors? for details.

The `--color` option will highlight matching patterns, line numbers, filename, etc. It has three different settings:

- `auto` will result in color highlighting when results are displayed on terminal, but not when 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 output is redirected to another command, file, etc
- `never` explicitly disable color highlighting

```
$ grep --color=auto -m3 'in' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
by definition, not smart enough to debug it by Brian W. Kernighan
Some people, when confronted with a problem, think - I know, I will
$ grep --color=auto -n -e '1' -e 'worth' *.txt
programming_quotes.txt:9:is not worth knowing by Alan Perlis
programming_quotes.txt:12:naming things, and off-by-1 errors by Leon Bambrick
search_strings.txt:2:1
$
```

Below image shows difference between `auto` and `always`. In the first case, `in` is highlighted even after piping, while in the second case, `in` is not highlighted. In practice, `always` is rarely used as it has extra information added to matching lines and could cause undesirable results when processing such lines.

```
$ grep --color=always 'in' programming_quotes.txt | grep --color 'not'
by definition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
$ grep --color=auto 'in' programming_quotes.txt | grep --color 'not'
by definition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
$
```

Usually, both `ls` and `grep` commands are aliased to include `--color=auto`.

```
$ # this is usually saved in ~/.bashrc or ~/.bash_aliases
$ alias grep='grep --color=auto'

$ # another use case for 'always' is piping the results to 'less' command
$ grep --color=always 'not' programming_quotes.txt | less -R
```

## Match whole word or line

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

The `-w` option behaves a bit differently than word boundaries in regular expressions. See Word boundary differences section for details.

```
$ # 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
```

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 if no other characters are present
$ printf 'see my book list\nmy book\n' | grep -x 'my book'
my book

$ grep '1' *.txt
programming_quotes.txt:naming things, and off-by-1 errors by Leon Bambrick
search_strings.txt:1
$ grep -x '1' *.txt
search_strings.txt:1

$ # counting empty lines, won't work for files with DOS style line endings
$ grep -cx '' programming_quotes.txt
3
```

## 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. In these cases, it is advised to use `-F` 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 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. This option is more commonly used with regular expressions.

```
$ grep -o -e 'twice' -e 'hard' programming_quotes.txt
twice
hard
hard

$ # -c only gives count of matching lines
$ grep -c 'in' programming_quotes.txt
8
$ # use -o to get each match on a separate line
$ grep -o 'in' programming_quotes.txt | wc -l
13
```

## Summary

In my initial years of cli usage as a VLSI engineer, I knew may be about five of the options listed in this chapter. Didn't even know about the `color` option. I've seen comments about not knowing `-c` option. These are some of the reasons why I'd advice to go through list of all the options if you are using a command frequently. Bonus points for maintaining a list of example usage for future reference, passing on to your colleagues, etc.

## Exercises

Create `exercises` directory and within it, create another directory for this chapter, say `freq_options` or `chapter_2` . Input is a file downloaded from internet — https://www.gutenberg.org/files/345/345.txt saved as `dracula.txt` . To solve the exercises, modify the partial command shown just before the expected output.

**a)** Display all lines containing `ablaze`

```
$ mkdir -p exercises/freq_options && cd $_
$ wget https://www.gutenberg.org/files/345/345.txt -O dracula.txt
```

```
$ grep ##### add your solution here
the room, his face all ablaze with excitement. He rushed up to me and
```

**b)** Display all lines containing `abandon` as a whole word.

```
$ grep ##### add your solution here
inheritors, being remote, would not be likely to abandon their just
```

**c)** Display all lines that satisfies both of these conditions:

- `professor` matched irrespective of case
- either `quip` or `sleep` matched case sensitively

```
$ grep ##### add your solution here
equipment of a professor of the healing craft. When we were shown in,
its potency; and she fell into a deep sleep. When the Professor was
sleeping, and the Professor seemingly had not moved from his seat at her
to sleep, and something weaker when she woke from it. The Professor and
```

**d)** Display first three lines containing `Count`

```
$ grep ##### add your solution here
town named by Count Dracula, is a fairly well-known place. I shall enter
must ask the Count all about them.)
Count Dracula had directed me to go to the Golden Krone Hotel, which I
```

**e)** Display first six lines containing `Harker` but not either of `Journal` or `Letter`

```
$ grep ##### add your solution here
said, "The Herr Englishman?" "Yes," I said, "Jonathan Harker." She
"I am Dracula; and I bid you welcome, Mr. Harker, to my house. Come in;
I shall be all alone, and my friend Harker Jonathan--nay, pardon me, I
Jonathan Harker will not be by my side to correct and aid me. He will be
"I write by desire of Mr. Jonathan Harker, who is himself not strong
junior partner of the important firm Hawkins & Harker; and so, as you
```

**f)** Display lines containing `Zooelogical Gardens` along with line number prefix.

```
$ grep ##### add your solution here
5597:         _Interview with the Keeper in the Zooelogical Gardens._
5601:the keeper of the section of the Zooelogical Gardens in which the wolf
8042:the Zooelogical Gardens a young one may have got loose, or one be bred
```

**g)** Find total count of whole word `the` (irrespective of case).

```
$ grep ##### add your solution here
8090
```

**h)** The below code snippet tries to get number of empty lines, but apparently shows wrong result, why?

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

# BRE/ERE Regular Expressions

This chapter will cover Basic and Extended Regular Expressions as implemented in `GNU grep`. Though not strictly conforming to POSIX specifications, most of it is applicable to other `grep` implementations as well. Unless otherwise indicated, examples and descriptions will assume ASCII input. `GNU grep` also supports Perl Compatible Regular Expressions, which will be covered in a later chapter.

By default, `grep` treats the search pattern as Basic Regular Expression (BRE)

- `-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)


Files used in examples are available chapter wise from learn_gnugrep_ripgrep repo. The directory for this chapter is `bre_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 Escaping metacharacters section).

There are two line anchors:

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

```
$ # lines starting with 'pa'
$ printf 'spared no one\npar\nspar\ndare' | grep '^pa'
par

$ # lines ending with 'ar'
$ printf 'spared no one\npar\nspar\ndare' | grep 'ar$'
par
spar

$ # lines containing only 'par'
$ 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 start or end of word.

The escape sequence `\b` denotes a word boundary. This works for both start of word and 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 boundary without a word character.

As an alternate, you can use `\<` to indicate start of word anchor and `\>` to indicate 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.

Word boundaries behave a bit differently than `-w` option. See Word boundary differences section for details.

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

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

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

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

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' word_anchors.txt
apparent effort
```

```
two spare computers

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

$ # match 'par' but not as end of word
$ grep 'par\B' word_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 expression separated by `|` is satisfied. These can have their own independent anchors as well.

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 `\?`, `\+`, `\{`, `\|`, `\(`, and `\)`.

```
$ # three different ways to match either 'cat' or 'dog'
$ printf 'I like cats\nI like parrots\nI like dogs' | grep 'cat\|dog'
I like cats
I like dogs
$ printf 'I like cats\nI like parrots\nI like dogs' | grep -E 'cat|dog'
I like cats
I like dogs
$ printf 'I like cats\nI like parrots\nI like dogs' | grep -e 'cat' -e 'dog'
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
$ # lines starting with 'a' or words ending with 'e'
$ grep -E '^a|e\b' word_anchors.txt
apparent effort
two spare computers
```

A cool use case of alternation is combining line anchors to display entire input file but highlight only required search patterns. Standalone line anchors will match every input line, even empty lines as they are position markers.

```
$ grep --color=auto -E '^|pare' word_anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart
$ grep --color=auto -E 'sub|put|tar|$' word_anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart
$
```

There's some tricky situations 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.

```
$ 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\narrest' | grep -E 'reform|rest'
reform
arrest
$ # with grouping
$ printf 'red\nreform\nread\narrest' | grep -E 're(form|st)'
reform
arrest

$ # without grouping
$ printf 'sub par\nspare\npart time' | grep -E '\bpar\b|\bpart\b'
sub par
part time
$ # taking out common anchors
$ printf 'sub par\nspare\npart time' | grep -E '\b(par|part)\b'
sub par
part time
$ # taking out common characters as well
$ # you'll later learn a better technique instead of using empty alternate
$ printf 'sub par\nspare\npart time' | grep -E '\bpar(|t)\b'
sub par
part time
```

## Escaping metacharacters

You have seen a few metacharacters and escape sequences that help to 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.

If there are many metacharacters to be escaped, try to work out if the command can be simplified 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
$ 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 at customary positions
```

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

$ # BRE vs 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` (commonly used to represent tab character). Neither does it support formats like `\xNN` (specifying a character by its ASCII value in hexadecimal format). As an alternate, you can use `bash` ANSI-C Quoting feature to use such escape sequences.

```
$ # any undefined escape sequence is treated as the character it escapes
$ # here \t is same as t
$ echo 'attempt' | grep -o 'a\tt'
att

$ # here $'..' is a bash feature to enable use of escape sequences
$ printf 'go\tto\ngo to' | grep $'go\tto'
go      to

$ # \x20 is hexadecimal for space character
$ printf 'go\tto\ngo to' | grep $'go\x20to'
go to
```

## The dot meta character

The dot metacharacter serves as a placeholder to match any character. Later you'll learn how to define your own custom placeholder for 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 `/usr/share/dict/words` dictionary file. This will be used as input file to illustrate regular expression examples. It is included in the repo as `words.txt` file (modified to make it ASCII only).

```
$ # 5 character lines starting with 'c' and ending with 'ty' or 'ly'
$ grep -xE 'c..(t|l)y' words.txt
catty
coyly
curly
```

## Quantifiers

As an analogy, alternation provides logical OR. Combining the dot metacharacter `.` and quantifiers (and alternation if needed) paves a way to perform logical AND. For example, to check if a string matches two patterns with any number of characters in between. Quantifiers can be applied to both characters and groupings. Apart from 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 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. There is no upper bound, more details will be discussed in the next section.

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

$ # extract zero or more of '1' followed by '2'
$ echo '3111111111125111142' | grep -o '1*2'
11111111112
2
```

The `+` metacharacter quantifies a character or group to match `1` or more times. Similar to `*` quantifier, there is no upper bound.

```
$ # 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
feeeed

$ # 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
feeeed

$ # extract one or more of '1' followed by '2'
$ echo '3111111111125111142' | grep -oE '1+2'
11111111112
$ # extract one or more of '1' followed by optional '4' and then '2'
$ echo '3111111111125111142' | grep -oE '1+4?2'
11111111112
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:

| Pattern | Description |
|---|---|
| `{m,n}` | match `m` to `n` times |
| `{m,}` | match at least `m` times |
| `{,n}` | match up to `n` times (including `0` times) |
| `{n}` | match exactly `n` times |

```
$ # note that space is not allowed after ,
$ # 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.

Next up, how to construct 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 'a cat and a dog' | grep -E 'cat.*dog|dog.*cat'
a cat and a dog
$ echo 'dog and cat' | grep -E 'cat.*dog|dog.*cat'
dog and cat
```

## Longest match wins

You've already seen an example with alternation, where the longest matching portion was chosen if two alternatives started from 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 '12baz' can match as well
$ echo 'foo123312baz' | grep -oE 'o(1|2|3)+(12baz)?'
o123312baz
$ # in other implementations like PCRE, that is not the case
$ # precedence is left to right for greedy quantifiers
$ echo 'foo123312baz' | grep -oP 'o(1|2|3)+(12baz)?'
o123312
```

While determining the longest match, 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 first `a` in the input string to the last `b` in the string. In other implementations, like PCRE, this is achieved through a process called **backtracking**. Both approaches have their own advantages and disadvantages and have cases where the pattern can result in exponential time consumption.

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

$ # extract from first 'c' to 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 first 'c' to 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\ncoat\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 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 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 design.

```
$ # 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 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 start of string based on a delimiter
$ echo 'foo=42; baz=123' | grep -o '^[^=]*'
foo

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

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

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

```
$ # 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
```

Some commonly used character sets have predefined escape sequences:

- `\w` matches all **word** characters `[a-zA-Z0-9_]` (recall the description for `-w` option)
- `\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 (but PCRE allows this). 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

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

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

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

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

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

$ printf 'err_msg\nxerox\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
```

Specific placement is needed to match character class metacharacters literally.

> ⚠ 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.

```
$ # - should be first or last character within []
$ echo 'ab-cd gh-c 12-423' | grep -owE '[a-z-]{2,}'
ab-cd
gh-c

$ # ] should be first character within []
$ printf 'int a[5]\nfoo\n1+1=2\n' | grep '[=]]'
$ printf 'int a[5]\nfoo\n1+1=2\n' | grep '[]=]'
int a[5]
1+1=2

$ # to match [ use [ anywhere in the character set
$ # but not combinations like [. or [:
$ # [][] 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 first character within []
$ echo 'f*(a^b) - 3*(a+b)/(a-b)' | grep -o 'a[+^]b'
a^b
a+b

$ # characters like \ and $ are not special within []
$ echo '5ba\babc2' | grep -o '[a\b]*'
ba\bab
```

## Backreferences

The grouping metacharacters `()` are also known as **capture groups**. Similar to variables in programming languages, the string captured by `()` can be referred later using backreference `\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`.

> ⚠ 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.

```
$ # 8 character lines having same 3 lowercase letters at 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

$ # same word next to each other
$ # 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
```

## Known Bugs

Visit grep bug list for known issues.

From `man grep` under **Known Bugs** section:

> 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 are very slow, and may require exponential time.

Here's is 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 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 repetition 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 problem
$ # 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}'
$ echo 'it line with it here it too sit' | grep -oP 'with(.*\bit\b){2}'
with it here it
```

## Summary

Knowing regular expressions very well is not only important to use `grep` effectively, but also comes in handy when moving onto use regular expressions in other 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 five variations — BRE, ERE, PCRE, Rust regex and PCRE2. However, core concepts are likely to be same and having a handy reference sheet would go a long way in reducing misuse.

## Exercises

**a)** Extract all pairs of `()` with/without text inside them, provided they do not contain `()` characters inside.

```
$ echo 'I got (12) apples' | grep ##### add your solution here
(12)

$ echo '((2 +3)*5)=25 and (4.3/2*()' | grep ##### add your solution here
(2 +3)
()
```

**b)** For the given input, match all lines that start with `den` or end with `ly`

```
$ lines='lovely\n1 dentist\n2 lonely\neden\nfly away\ndent'
$ printf '%b' "$lines" | grep ##### add your solution here
lovely
2 lonely
dent
```

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

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

```
nice1423
cool_42a
```

**d)** Each line in given input contains a single word. Match all lines containing `car` but not as a whole word.

```
$ printf 'car\nscar\ncare\npot\nscare\n' | grep ##### add your solution here
scar
care
scare
```

**e)** For `dracula.txt` file, count the total number of lines that contain `removed` or `rested` or `received` or `replied` or `refused` or `retired` as whole words.

```
$ grep ##### add your solution here
73
```

**f)** Extract words starting with `s` and containing `e` and `t` in any order.

```
$ words='sequoia subtle exhibit sets tests sit'
$ echo "$words" | grep ##### add your solution here
subtle
sets
```

**g)** Extract all whole words having the same first and last character.

```
$ echo 'oreo not a pip roar took 22' | grep ##### add your solution here
oreo
a
pip
roar
22
```

**h)** Match all lines containing `*[5]`

```
$ printf '4*5]\n(9-2)*[5]\n[5]*3\nr*[5\n' | grep ##### add your solution here
(9-2)*[5]
```

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

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

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

**k)** `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
```

**l)** Display all lines that start with `hand` and ends with no further character or `s` or `y` or `le` .

```
$ lines='handed\nhand\nhandy\nunhand\nhands\nhandle\n'
$ printf '%b' "$lines" | grep ##### add your solution here
hand
handy
hands
handle
```