learnbyexample

CLI text processing with GNU sed

✓ 200+ examples

✓ 50+ exercises

Sundeep Agarwal

Table of contents

Preface

You are likely to be familiar with using a search and replace dialog (usually invoked with the `Ctrl+H` shortcut) to locate the occurrences of a particular string and replace it with something else. The `sed` command is a versatile and feature-rich version for search and replace operations, 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.

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. However, commands to manipulate data buffers and multiline techniques will be discussed only briefly and some commands are skipped entirely.

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. Knowing the basics of the `grep` command will be handy in understanding the filtering features of `sed` .

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

Conventions

- The examples presented here have been tested with **GNU sed** version **4.9** and may include features not available in earlier versions.
- Code snippets are copy pasted from the `GNU bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines to improve readability, only `real` time shown for speed comparisons, output skipped for commands like `wget` and so on.
- Unless otherwise noted, all examples and explanations are meant for **ASCII** input.
- `sed` would mean `GNU sed` , `grep` would mean `GNU grep` and so on unless otherwise specified.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The learn_gnused repo has all the code snippets and files used in examples, exercises and other details related to the book. If you are not familiar with the `git` command, click the **Code** button on the webpage to get the files.

Acknowledgements

- GNU sed documentation — manual and examples
- stackoverflow and unix.stackexchange — for getting answers to pertinent questions on `sed` 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
- arifmahmudrana for spotting an ambiguous explanation

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_gnused/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep 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

Book version

2.1

See Version_changes.md to track changes across book versions.

Introduction

The command name `sed` is derived from **s**tream **ed**itor. Here, stream refers to data being passed via shell pipes. Thus, the command's primary functionality is to act as a text editor for **stdin** data with **stdout** as the output target. It is now common to use `sed` on file inputs as well as in-place file editing.

This chapter will show how to install the latest `sed` version followed by details related to documentation. Then, you'll get an introduction to the **substitute** command, which is the most commonly used feature. The chapters to follow will add more details to the substitute command, discuss other commands and command line options. Cheatsheet, summary and exercises are also included at the end of these chapters.

Installation

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

`GNU sed` is part of the text creation and manipulation tools and comes by default on GNU/Linux distributions. To install a particular version, visit gnu: sed 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.

```
$ wget https://ftp.gnu.org/gnu/sed/sed-4.9.tar.xz
$ tar -Jxf sed-4.9.tar.xz
$ cd sed-4.9/
# see https://askubuntu.com/q/237576 if you get compiler not found error
$ ./configure
$ make
$ sudo make install

$ sed --version | head -n1
sed (GNU sed) 4.9
```

If you are not using a Linux distribution, you may be able to access `GNU sed` 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)

Documentation

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

```
$ man sed
SED(1)                          User Commands                          SED(1)

```

```
NAME
       sed - stream editor for filtering and transforming text

SYNOPSIS
       sed [-V] [--version] [--help] [-n] [--quiet] [--silent]
           [-l N] [--line-length=N] [-u] [--unbuffered]
           [-E] [-r] [--regexp-extended]
           [-e script] [--expression=script]
           [-f script-file] [--file=script-file]
           [script-if-no-other-script]
           [file...]

DESCRIPTION
       Sed  is  a  stream  editor.  A stream editor is used to perform basic
       text transformations on an input stream (a file or input from a pipe-
       line).   While  in  some  ways  similar  to  an  editor which permits
       scripted edits (such as ed), sed works by making only one  pass  over
       the  input(s),  and  is consequently more efficient.  But it is sed's
       ability to filter text in a pipeline which particularly distinguishes
       it from other types of editors.
```

Options overview

For a quick overview of all the available options, use `sed --help` from the command line.

```
# only partial output shown here
$ sed --help
  -n, --quiet, --silent
                 suppress automatic printing of pattern space
      --debug
                 annotate program execution
  -e script, --expression=script
                 add the script to the commands to be executed
  -f script-file, --file=script-file
                 add the contents of script-file to the commands to be executed
  --follow-symlinks
                 follow symlinks when processing in place
  -i[SUFFIX], --in-place[=SUFFIX]
                 edit files in place (makes backup if SUFFIX supplied)
  -l N, --line-length=N
                 specify the desired line-wrap length for the 'l' command
  --posix
                 disable all GNU extensions.
  -E, -r, --regexp-extended
                 use extended regular expressions in the script
                 (for portability use POSIX -E).
  -s, --separate
                 consider files as separate rather than as a single,
                 continuous long stream.
      --sandbox
                 operate in sandbox mode (disable e/r/w commands).
  -u, --unbuffered
                 load minimal amounts of data from the input files and flush
                 the output buffers more often
  -z, --null-data
                 separate lines by NUL characters
      --help     display this help and exit
      --version  output version information and exit

If no -e, --expression, -f, or --file option is given, then the first
non-option argument is taken as the sed script to interpret.  All
remaining arguments are names of input files; if no input files are
specified, then the standard input is read.
```

Editing standard input

`sed` has various commands to manipulate text. The **substitute** command is the most commonly used operation, which helps to replace matching portions with something else. The syntax is `s/REGEXP/REPLACEMENT/FLAGS` where:

- `s` stands for the **substitute** command
- `/` is an idiomatic delimiter character to separate various portions of the command
- `REGEXP` stands for **regular expression**
- `REPLACEMENT` refers to the replacement string
- `FLAGS` are options to change the default behavior of the command

For now, it is enough to know that the `s` command is used for search and replace operations.

```
# input data that'll be passed to sed for editing
$ printf '1,2,3,4\na,b,c,d\n'
1,2,3,4
a,b,c,d

# for each input line, change only the first ',' to '-'
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/'
1-2,3,4
a-b,c,d

# change all matches by adding the 'g' flag
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/g'
1-2-3-4
a-b-c-d
```

In the above example, the input data is created using the `printf` command to showcase stream editing. By default, `sed` processes the input line by line. The newline character `\n` is the line separator by default. The first `sed` command replaces only the first occurrence of `,` with `-` . The second command replaces all occurrences as the `g` flag is also used (`g` stands for `global`).

As a good practice, always use single quotes around the script argument. Examples requiring shell interpretation will be discussed later.

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
```

Editing file input

Although `sed` derives its name from *stream editing*, it is common to use `sed` for file editing. To do so, you can pass one or more input filenames as arguments. You can use `-` to represent stdin as one of the input sources. By default, the modified data will go to the `stdout` stream and the input files are not modified. In-place file editing chapter will discuss how to apply the changes back to the source files.

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

```
$ cat greeting.txt
Hi there
Have a nice day

# for each line, change the first occurrence of 'day' with 'weekend'
$ sed 's/day/weekend/' greeting.txt
Hi there
Have a nice weekend

# change all occurrences of 'e' to 'E'
# redirect the modified data to another file
$ sed 's/e/E/g' greeting.txt > out.txt
$ cat out.txt
Hi thErE
HavE a nicE day
```

In the previously seen examples, every input line had matched the search expression. The first `sed` command here searched for `day`, which did not match the first line of `greeting.txt` file. By default, even if a line doesn't satisfy the search expression, it will be part of the output. You'll see how to get only the modified lines in the Print command section.

Cheatsheet and summary

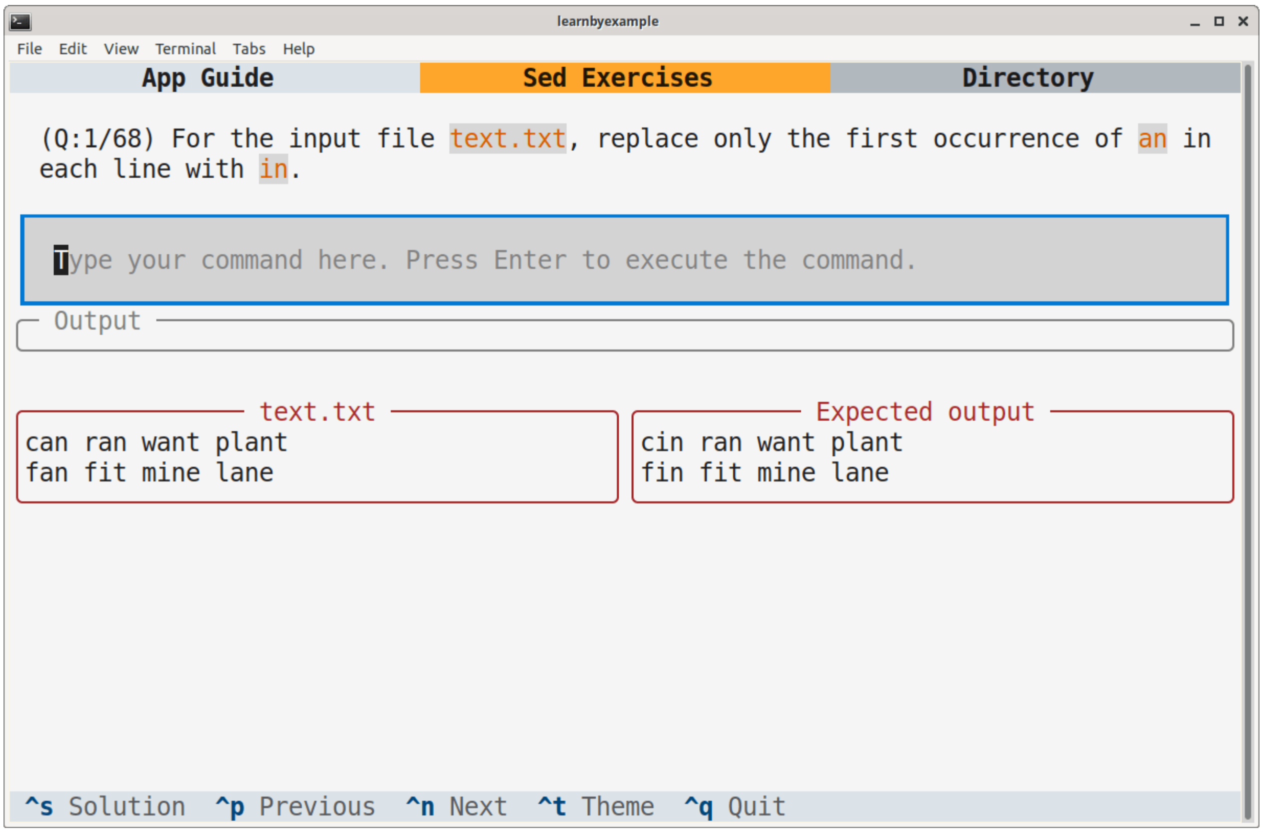
Note	Description
`man sed`	brief manual
`sed --help`	brief description of all the command line options
`info sed`	comprehensive manual
online gnu sed manual	well formatted, easier to read and navigate
`s/REGEXP/REPLACEMENT/FLAGS`	syntax for the substitute command
`sed 's/,/-/'`	for each line, replace the first `,` with `-`
`sed 's/,/-/g'`	replace all `,` with `-`

This introductory chapter covered installation process, documentation and how to search and replace text using `sed` from the command line. In the coming chapters, you'll learn many more commands and features that make `sed` an important tool when it comes to command line text processing. One such feature is editing files in-place, which will be discussed in the next chapter.

Interactive exercises

I wrote a TUI app to help you solve some of the exercises from this book interactively. See SedExercises repo for installation steps and app_guide.md for instructions on using this app.

Here's a sample screenshot:



```
(Q:1/68) For the input file text.txt, replace only the first occurrence of an in
each line with in.

Type your command here. Press Enter to execute the command.

Output

text.txt
can ran want plant
fan fit mine lane

Expected output
cin ran want plant
fin fit mine lane

^s Solution  ^p Previous  ^n Next  ^t Theme  ^q Quit
```

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) Replace only the first occurrence of `5` with `five` for the given stdin source.

```
$ echo 'They ate 5 apples and 5 mangoes' | sed ##### add your solution here
They ate five apples and 5 mangoes
```

2) Replace all occurrences of `5` with `five` .

```
$ echo 'They ate 5 apples and 5 mangoes' | sed ##### add your solution here
They ate five apples and five mangoes
```

3) Replace all occurrences of `0xA0` with `0x50` and `0xFF` with `0x7F` for the given input file.

```
$ cat hex.txt
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0

$ sed ##### add your solution here
start address: 0x50, func1 address: 0x50
end address: 0x7F, func2 address: 0xB0
```

4) The substitute command searches and replaces sequences of characters. When you need to map one or more characters with another set of corresponding characters, you can use the `y` command. Quoting from the manual:

y/src/dst/ Transliterate any characters in the pattern space which match any of the source-chars with the corresponding character in dest-chars.

Use the `y` command to transform the given input string to get the output string as shown below.

```
$ echo 'goal new user sit eat dinner' | sed ##### add your solution here
gOAl nEw UsEr sIt EAt dInnEr
```

5) Why does the following command produce an error? How'd you fix it?

```
$ echo 'a sunny day' | sed s/sunny day/cloudy day/
sed: -e expression #1, char 7: unterminated `s' command

# expected output
$ echo 'a sunny day' | sed ##### add your solution here
a cloudy day
```

In-place file editing

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

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

With backup

When an extension is provided as an argument to the `-i` option, the original contents of the input file gets preserved as per the extension given. For example, if the input file is `ip.txt` and `-i.orig` is used, the backup file will be named as `ip.txt.orig`.

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

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

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

Without backup

Sometimes backups are not desirable. In such cases, you can use the `-i` option without an argument. Be careful though, as changes made cannot be undone. It is recommended to test the command with sample inputs before applying the `-i` option on the actual file. You could also use the option with backup, compare the differences with a `diff` program and then delete the backup.

```
$ cat fruits.txt
banana
papaya
mango
```

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

Multiple files

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

```
$ cat f1.txt
have a nice day
bad morning
what a pleasant evening
$ cat f2.txt
worse than ever
too bad

$ sed -i.bkp 's/bad/good/' f1.txt f2.txt
$ ls f?.*
f1.txt  f1.txt.bkp  f2.txt  f2.txt.bkp

$ cat f1.txt
have a nice day
good morning
what a pleasant evening
$ cat f2.txt
worse than ever
too good
```

Prefix backup name

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

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

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

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

Place backups in a different directory

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

```
$ mkdir backups
$ sed -i'backups/*' 's/good/nice/' f1.txt f2.txt
$ ls backups/
f1.txt  f2.txt
```

Cheatsheet and summary

Note	Description
`-i`	after processing, write back changes to the source files changes made cannot be undone, so use this option with caution
`-i.bkp`	in addition to in-place editing, preserve original contents to a file whose name is derived from the input filename and `.bkp` as a suffix
`-i'bkp.*'`	`*` here gets replaced with the input filename thus providing a way to add a prefix instead of a suffix
`-i'backups/*'`	this will place the backup copy in a different existing directory instead of the source directory

This chapter discussed about the `-i` option which is useful when you need to edit a file in-place. This is particularly useful in automation scripts. But, do ensure that you have tested the `sed` command before applying to actual files if you need to use this option without creating backups. In the next chapter, you'll learn filtering features of `sed` and how that helps to apply commands to only certain input lines instead of all the lines.

Exercises

The exercises directory has all the files used in this section.

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

```
$ cat text.txt
can ran want plant
tin fin fit mine line
$ sed ##### add your solution here

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

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

```
$ cat text.txt
can ran want plant
tan fan fit mane lane
$ sed ##### add your solution here

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

3) For the input file `copyright.txt`, replace `copyright: 2018` with `copyright: 2019` and write back the changes to `copyright.txt` itself. The original contents should get saved to `2018_copyright.txt.bkp`

```
$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
$ sed ##### add your solution here

$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2019
$ cat 2018_copyright.txt.bkp
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
```

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

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

5) For the input file `pets.txt`, remove the first occurrence of `I like ` from each line and write back the changes to `pets.txt` itself. The original contents should get saved with the same filename inside the `bkp` directory. Assume that you do not know whether `bkp` exists or not in the current working directory.

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

##### add your solution here

$ cat pets.txt
cats
parrots
dogs
$ cat bkp/pets.txt
I like cats
I like parrots
I like dogs
```

Selective editing

By default, `sed` acts on the entire input content. Many a times, you want to act only upon specific portions of the input. To that end, `sed` has features to filter lines, similar to tools like `grep` , `head` and `tail` . `sed` can replicate most of `grep` 's filtering features without too much fuss. And has additional features like line number based filtering, selecting lines between two patterns, relative addressing, etc. If you are familiar with functional programming, you would have come across the **map, filter, reduce** paradigm. A typical task with `sed` involves filtering a subset of input and then modifying (mapping) them. Sometimes, the subset is the entire input, as seen in the examples of previous chapters.

A tool optimized for a particular functionality should be preferred where possible. `grep` , `head` and `tail` would be better performance wise compared to `sed` for equivalent line filtering solutions.

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

REGEXP filtering

As seen earlier, syntax for the substitute command is `s/REGEXP/REPLACEMENT/FLAGS` . The `/REGEXP/FLAGS` portion can be used as a conditional expression to allow commands to execute only for the lines matching the pattern.

```
# change commas to hyphens only if the input line contains '2'
# space between the filter and the command is optional
$ printf '1,2,3,4\na,b,c,d\n' | sed '/2/ s/,/-/g'
1-2-3-4
a,b,c,d
```

Use `/REGEXP/FLAGS!` to act upon lines other than the matching ones.

```
# change commas to hyphens if the input line does NOT contain '2'
# space around ! is optional
$ printf '1,2,3,4\na,b,c,d\n' | sed '/2/! s/,/-/g'
1,2,3,4
a-b-c-d
```

`/REGEXP/` is one of the ways to define a filter, termed as **address** in the manual. Others will be covered later in this chapter.

Regular expressions will be discussed later. In this chapter, the examples with `/REGEXP/` filtering will use only fixed strings (exact string comparison).

Delete command

To **d**elete the filtered lines, use the `d` command. Recall that all input lines are printed by default.

```
# same as: grep -v 'at'
$ printf 'sea\neat\ndrop\n' | sed '/at/d'
sea
drop
```

To get the default `grep` filtering, use the `!d` combination. Sometimes, negative logic can get confusing to use. It boils down to personal preference, similar to choosing between `if` and `unless` conditionals in programming languages.

```
# same as: grep 'at'
$ printf 'sea\neat\ndrop\n' | sed '/at/!d'
eat
```

Using an **address** is optional. So, for example, `sed '!d' file` would be equivalent to the `cat file` command.

```
# same as: cat greeting.txt
$ sed '!d' greeting.txt
Hi there
Have a nice day
```

Print command

To **p**rint the filtered lines, use the `p` command. But, recall that all input lines are printed by default. So, this command is typically used in combination with the `-n` option, which turns off the default printing.

```
$ cat rhymes.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

# same as: grep 'warm' rhymes.txt
$ sed -n '/warm/p' rhymes.txt
it is a warm and cozy day

# same as: grep 'n t' rhymes.txt
$ sed -n '/n t/p' rhymes.txt
listen to what I say
go play in the park
```

The substitute command provides `p` as a flag. In such a case, the modified line would be printed only if the substitution succeeded.

```
$ sed -n 's/warm/cool/gp' rhymes.txt
it is a cool and cozy day

# filter + substitution + p combination
$ sed -n '/the/ s/ark/ARK/gp' rhymes.txt
go play in the pARK
come back before the sky turns dARK
```

Using `!p` with the `-n` option will be equivalent to using the `d` command.

```
# same as: sed '/at/d'
$ printf 'sea\neat\ndrop\n' | sed -n '/at/!p'
sea
drop
```

Here's an example of using the `p` command without the `-n` option.

```
# duplicate every line
$ seq 2 | sed 'p'
1
1
2
2
```

Quit commands

The `q` command causes `sed` to exit immediately. Remaining commands and input lines will not be processed.

```
# quits after an input line containing 'say' is found
$ sed '/say/q' rhymes.txt
it is a warm and cozy day
listen to what I say
```

The `Q` command is similar to `q` but won't print the matching line.

```
# matching line won't be printed
$ sed '/say/Q' rhymes.txt
it is a warm and cozy day
```

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

```
$ tac rhymes.txt | sed '/an/q' | tac
Bread, Butter and Jelly
Try them all before you perish
```

You can optionally provide an exit status (from `0` to `255`) along with the quit commands.

```
$ printf 'sea\neat\ndrop\n' | sed '/at/q2'
sea
eat
$ echo $?
2
```

```
$ printf 'sea\neat\ndrop\n' | sed '/at/Q3'
sea
$ echo $?
3
```

⚠ Be careful if you want to use `q` or `Q` commands with multiple files, as `sed` will stop even if there are other files remaining to be processed. You could use a mixed address range as a workaround. See also unix.stackexchange: applying q to multiple files.

Multiple commands

Commands seen so far can be specified more than once by separating them using `;` or using the `-e` option multiple times. See sed manual: Multiple commands syntax for more details.

```
# print all input lines as well as the modified lines
$ printf 'sea\neat\ndrop\n' | sed -n -e 'p' -e 's/at/AT/p'
sea
eat
eAT
drop

# equivalent command to the above example using ; instead of -e
# space around ; is optional
$ printf 'sea\neat\ndrop\n' | sed -n 'p; s/at/AT/p'
sea
eat
eAT
drop
```

You can also separate the commands using a literal newline character. If many lines are needed, it is better to use a sed script instead.

```
# here, each command is separated by a literal newline character
# similar to $ representing the primary prompt PS1,
# > represents the secondary prompt PS2
$ sed -n '
> /the/ s/ark/ARK/gp
> s/warm/cool/gp
> s/Bread/Cake/gp
> ' rhymes.txt
it is a cool and cozy day
go play in the pARK
come back before the sky turns dARK
Cake, Butter and Jelly
```

⚠ Do not use multiple commands to construct conditional OR of multiple search strings, as you might get lines duplicated in the output as shown below. You can use regular expression feature alternation for such cases.

```
$ sed -ne '/play/p' -e '/ark/p' rhymes.txt
go play in the park
go play in the park
come back before the sky turns dark
```

To execute multiple commands for a common filter, use `{}` to group the commands. You can also nest them if needed.

```
# spaces around {} is optional
$ printf 'gates\nnot\nused\n' | sed '/e/{s/s/*/g; s/t/*/g}'
ga*e*
not
u*ed

$ sed -n '/the/{s/for/FOR/gp; /play/{p; s/park/PARK/gp}}' rhymes.txt
go play in the park
go play in the PARK
come back beFORe the sky turns dark
Try them all beFORe you perish
```

Command grouping is an easy way to construct conditional AND of multiple search strings.

```
# same as: grep 'ark' rhymes.txt | grep 'play'
$ sed -n '/ark/{/play/p}' rhymes.txt
go play in the park

# same as: grep 'the' rhymes.txt | grep 'for' | grep 'urn'
$ sed -n '/the/{/for/{/urn/p}}' rhymes.txt
come back before the sky turns dark

# same as: grep 'for' rhymes.txt | grep -v 'sky'
$ sed -n '/for/{/sky/!p}' rhymes.txt
Try them all before you perish
```

Other solutions using alternation feature of regular expressions and `sed` 's control structures will be discussed later.

Line addressing

Line numbers can also be used as a filtering criteria.

```
# here, 3 represents the address for the print command
# same as: head -n3 rhymes.txt | tail -n1
# same as: sed '3!d'
$ sed -n '3p' rhymes.txt
go play in the park
```

```
# print the 2nd and 6th lines
$ sed -n '2p; 6p' rhymes.txt
listen to what I say
There are so many delights to cherish

# apply substitution only for the 2nd line
$ printf 'gates\nnot\nused\n' | sed '2 s/t/*/g'
gates
no*
used
```

As a special case, `$` indicates the last line of the input.

```
# same as: tail -n1 rhymes.txt
$ sed -n '$p' rhymes.txt
Try them all before you perish
```

For large input files, use the `q` command to avoid processing unnecessary input lines.

```
$ seq 3542 4623452 | sed -n '2452{p; q}'
5993
$ seq 3542 4623452 | sed -n '250p; 2452{p; q}'
3791
5993

# here is a sample time comparison
$ time seq 3542 4623452 | sed -n '2452{p; q}' > f1
real    0m0.005s
$ time seq 3542 4623452 | sed -n '2452p' > f2
real    0m0.121s
$ rm f1 f2
```

Mimicking the `head` command using line number addressing and the `q` command.

```
# same as: seq 23 45 | head -n5
$ seq 23 45 | sed '5q'
23
24
25
26
27
```

Print only the line number

The `=` command will display the line numbers of matching lines.

```
# gives both the line number and matching lines
$ grep -n 'the' rhymes.txt
3:go play in the park
4:come back before the sky turns dark
9:Try them all before you perish

```

```
# gives only the line number of matching lines
# note the use of the -n option to avoid default printing
$ sed -n '/the/=' rhymes.txt
3
4
9
```

If needed, matching line can also be printed. But there will be a newline character between the matching line and the line number.

```
$ sed -n '/what/{=; p}' rhymes.txt
2
listen to what I say

$ sed -n '/what/{p; =}' rhymes.txt
listen to what I say
2
```

Address range

So far, filtering has been based on specific line number or lines matching the given REGEXP pattern. Address range gives the ability to define a starting address and an ending address separated by a comma.

```
# note that all the matching ranges are printed
$ sed -n '/to/,/pl/p' rhymes.txt
listen to what I say
go play in the park
There are so many delights to cherish
Apple, Banana and Cherry

# same as: sed -n '3,8!p'
$ seq 15 24 | sed '3,8d'
15
16
23
24
```

Line numbers and REGEXP filtering can be mixed.

```
$ sed -n '6,/utter/p' rhymes.txt
There are so many delights to cherish
Apple, Banana and Cherry
Bread, Butter and Jelly

# same as: sed '/play/Q' rhymes.txt
# inefficient, but this will work for multiple file inputs
$ sed '/play/,$d' rhymes.txt
it is a warm and cozy day
listen to what I say
```

If the second filtering condition doesn't match, lines starting from the first condition to the last line of the input will be matched.

```
# there's a line containing 'Banana' but the matching pair isn't found
$ sed -n '/Banana/,/XYZ/p' rhymes.txt
Apple, Banana and Cherry
Bread, Butter and Jelly
Try them all before you perish
```

The second address will always be used as a filtering condition only from the line that comes after the line that satisfied the first address. For example, if the same search pattern is used for both the addresses, there'll be at least two lines in output (assuming there are lines in the input after the first matching line).

```
$ sed -n '/w/,/w/p' rhymes.txt
it is a warm and cozy day
listen to what I say

# there's no line containing 'Cherry' after the 7th line
# so, rest of the file gets printed
$ sed -n '7,/Cherry/p' rhymes.txt
Apple, Banana and Cherry
Bread, Butter and Jelly
Try them all before you perish
```

As a special case, the first address can be `0` if the second one is a REGEXP filter. This allows the search pattern to be matched against the first line of the file.

```
# same as: sed '/cozy/q'
# inefficient, but this will work for multiple file inputs
$ sed -n '0,/cozy/p' rhymes.txt
it is a warm and cozy day

# same as: sed '/say/q'
$ sed -n '0,/say/p' rhymes.txt
it is a warm and cozy day
listen to what I say
```

Relative addressing

The `grep` command has an option `-A` that allows you to view lines that come *after* the matching lines. The `sed` command provides a similar feature when you prefix a `+` character to the number used in the second address. One difference compared to `grep` is that the context lines won't trigger a fresh matching of the first address.

```
# match a line containing 'the' and display the next line as well
# won't be same as: grep -A1 --no-group-separator 'the'
$ sed -n '/the/,+1p' rhymes.txt
go play in the park
come back before the sky turns dark
Try them all before you perish

# the first address can be a line number too
# helpful when it is programmatically constructed in a script
$ sed -n '6,+2p' rhymes.txt
There are so many delights to cherish
Apple, Banana and Cherry
Bread, Butter and Jelly
```

You can construct an arithmetic progression with start and step values separated by the `~` symbol. `i~j` will filter lines numbered `i+0j` , `i+1j` , `i+2j` , `i+3j` , etc. So, `1~2` means all odd numbered lines and `5~3` means 5th, 8th, 11th, etc.

```
# print even numbered lines
$ seq 10 | sed -n '2~2p'
2
4
6
8
10

# delete lines numbered 2+0*4, 2+1*4, 2+2*4, etc (2, 6, 10, etc)
$ seq 7 | sed '2~4d'
1
3
4
5
7
```

If `i,~j` is used (note the `,`) then the meaning changes completely. After the start address, the closest line number which is a multiple of `j` will mark the end address. The start address can be REGEXP based filtering as well.

```
# here, closest multiple of 4 is the 4th line
$ seq 10 | sed -n '2,~4p'
2
3
4
# here, closest multiple of 4 is the 8th line
$ seq 10 | sed -n '5,~4p'
5
6
7
8

# line matching 'many' is the 6th line, closest multiple of 3 is the 9th line
$ sed -n '/many/,~3p' rhymes.txt
There are so many delights to cherish
Apple, Banana and Cherry
Bread, Butter and Jelly
Try them all before you perish
```

n and N commands

So far, the commands used have all been processing only one line at a time. The address range option provides the ability to act upon a group of lines, but the commands still operate one line at a time for that group. There are cases when you want a command to handle a string that contains multiple lines. As mentioned in the preface, this book will not cover advanced commands related to multiline processing and I highly recommend using `awk` or `perl` for such scenarios. However, this section will introduce two commands `n` and `N` which are relatively easier to use and will be seen in the coming chapters as well.

This is also a good place to get to know more details about how `sed` works. Quoting from sed manual: How sed Works:

> sed maintains two data buffers: the active pattern space, and the auxiliary hold space. Both are initially empty.
> sed operates by performing the following cycle on each line of input: first, sed reads one line from the input stream, removes any trailing newline, and places it in the pattern space. Then commands are executed; each command can have an address associated to it: addresses are a kind of condition code, and a command is only executed if the condition is verified before the command is to be executed.
> When the end of the script is reached, unless the -n option is in use, the contents of pattern space are printed out to the output stream, adding back the trailing newline if it was removed. Then the next cycle starts for the next input line.

The **pattern space** buffer has only contained single line of input in all the examples seen so far. By using `n` and `N` commands, you can change the contents of the pattern space and use commands to act upon entire contents of this data buffer. For example, you can perform substitution on two or more lines at once.

First up, the `n` command. Quoting from sed manual: Often-Used Commands:

> If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then sed exits without processing any more commands.

```
# same as: sed -n '2~2p'
# n will replace pattern space with the next line of input
# as -n option is used, the replaced line won't be printed
# the p command then prints the new line
$ seq 10 | sed -n 'n; p'
2
4
6
8
10

# if a line contains 't', replace pattern space with the next line
# substitute all 't' with 'TTT' for the new line thus fetched
# note that 't' wasn't substituted in the line that got replaced
# replaced pattern space gets printed as -n option is NOT used here
$ printf 'gates\nnot\nused\n' | sed '/t/{n; s/t/TTT/g}'
```

```
gates
noTTT
used
```

Next, the `N` command. Quoting from sed manual: Less Frequently-Used Commands:

> Add a newline to the pattern space, then append the next line of input to the pattern space. If there is no more input then sed exits without processing any more commands. When -z is used, a zero byte (the ascii 'NUL' character) is added between the lines (instead of a new line).

```
# append the next line to the pattern space
# and then replace newline character with a colon character
$ seq 7 | sed 'N; s/\n/:/'
1:2
3:4
5:6
7

# if line contains 'at', the next line gets appended to the pattern space
# then the substitution is performed on the two lines in the buffer
$ printf 'gates\nnot\nused\n' | sed '/at/{N; s/s\nnot/d/}'
gated
used
```

> See also sed manual: N command on the last line. Escape sequences like `\n` will be discussed in detail later.

> See grymoire: sed tutorial if you wish to explore about the data buffers in detail and learn about the various multiline commands.

Cheatsheet and summary

Note	Description
`ADDR cmd`	Execute cmd only if the input line satisfies the ADDR condition
	`ADDR` can be REGEXP or line number or a combination of them
`/at/d`	delete all lines satisfying the given REGEXP
`/at/!d`	don't delete lines matching the given REGEXP
`/twice/p`	print all lines based on the given REGEXP
	as print is the default action, usually `p` is paired with `-n`
`/not/ s/in/out/gp`	substitute only if line matches the given REGEXP
	and print only if the substitution succeeds
`/if/q`	quit immediately after printing the current pattern space
	further input files, if any, won't be processed
`/if/Q`	quit immediately without printing the current pattern space
`/at/q2`	both `q` and `Q` can additionally use `0-255` as the exit code
`-e 'cmd1' -e 'cmd2'`	execute multiple commands one after the other

Note	Description
`cmd1; cmd2`	execute multiple commands one after the other
	note that not all commands can be constructed this way
	commands can also be separated by a literal newline character
`ADDR {cmds}`	group one or more commands to be executed for given ADDR
	groups can be nested as well
	ex: `/in/{/not/{/you/p}}` conditional AND of 3 REGEXPs
`2p`	line addressing, print only the 2nd line
`$`	special address to indicate the last line of input
`2452{p; q}`	quit early to avoid processing unnecessary lines
`/not/=`	print line number instead of the matching line
`ADDR1,ADDR2`	start and end addresses to operate upon
	if ADDR2 doesn't match, lines till end of the file gets processed
`/are/,/by/p`	print all groups of line matching the REGEXPs
`3,8d`	delete lines numbered 3 to 8
`5,/use/p`	line number and REGEXP can be mixed
`0,/not/p`	inefficient equivalent of `/not/q` but works for multiple files
`ADDR,+N`	all lines matching the ADDR and `N` lines after
`i~j`	arithmetic progression with `i` as start and `j` as step
`ADDR,~j`	closest multiple of `j` w.r.t. the line matching the ADDR
pattern space	active data buffer, commands work on this content
`n`	if `-n` option isn't used, pattern space gets printed
	and then pattern space is replaced with the next line of input
	exit without executing other commands if there's no more input
`N`	add newline (or NUL for `-z`) to the pattern space
	and then append the next line of input
	exit without executing other commands if there's no more input

This chapter introduced the filtering capabilities of `sed` and how it can be combined with `sed` commands to process only lines of interest instead of the entire input contents. Filtering can be specified using a REGEXP, line number or a combination of them. You also learnt various ways to compose multiple `sed` commands. In the next chapter, you will learn syntax and features of regular expressions as supported by the `sed` command.

Exercises

The exercises directory has all the files used in this section.

1) For the given input, display except the third line.

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

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

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

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

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

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

4) Use `sed` to get the output shown below for the given input. You'll have to first understand the input to output transformation logic and then use commands introduced in this chapter to construct a solution.

```
$ seq 15 | sed ##### add your solution here
2
4
7
9
12
14
```

5) For the input file `addr.txt`, display all lines from the start of the file till the first occurrence of `is`.

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

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

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

7) `n` and `N` commands will not execute further commands if there are no more input lines to fetch. Correct the command shown below to get the expected output.

```
# wrong output
$ seq 11 | sed 'N; N; s/\n/-/g'
1-2-3
```

```
4-5-6
7-8-9
10
11

# expected output
$ seq 11 | sed ##### add your solution here
1-2-3
4-5-6
7-8-9
10-11
```

8) For the input file `addr.txt` , add line numbers in the format as shown below.

```
$ sed ##### add your solution here
1
Hello World
2
How are you
3
This game is good
4
Today is sunny
5
12345
6
You are funny
```

9) For the input file `addr.txt` , print all lines that contain `are` and the line that comes after, if any.

```
$ sed ##### add your solution here
How are you
This game is good
You are funny
```

Bonus: For the above input file, will `sed -n '/is/,+1 p' addr.txt` produce identical results as `grep -A1 'is' addr.txt` ? If not, why?

10) Print all lines if their line numbers follow the sequence 1, 15, 29, 43, etc but not if the line contains `4` in it.

```
$ seq 32 100 | sed ##### add your solution here
32
60
88
```

11) For the input file `sample.txt` , display from the start of the file till the first occurrence of `are` , excluding the matching line.

```
$ cat sample.txt
Hello World

```

```
Hi there
How are you

Just do-it
Believe it

banana
papaya
mango

Much ado about nothing
He he he
Adios amigo

$ sed ##### add your solution here
Hello World

Hi there
```

12) For the input file `sample.txt` , display from the last occurrence of `do` till the end of the file.

```
##### add your solution here
Much ado about nothing
He he he
Adios amigo
```

13) For the input file `sample.txt` , display from the 9th line till a line containing `go` .

```
$ sed ##### add your solution here
banana
papaya
mango
```

14) For the input file `sample.txt` , display from a line containing `it` till the next line number that is divisible by 3.

```
$ sed ##### add your solution here
Just do-it
Believe it

banana
```

15) Display only the odd numbered lines from `addr.txt` .

```
$ sed ##### add your solution here
Hello World
This game is good
12345
```