learnbyexample

COMMAND LINE TEXT PROCESSING with GNU COREUTILS

an example based guide

Sundeep Agarwal

Table of contents

Preface

You might be already aware of popular coreutils commands like `head` , `tail` , `tr` , `sort` , etc. This book will teach you more than twenty of such specialized text processing tools provided by the `GNU coreutils` package.

My Command Line Text Processing repo includes chapters on some of these coreutils commands. Those chapters have been significantly edited for this book and new chapters have been added to cover more commands.

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 my curated resources on Linux CLI and Shell scripting before starting this book.

Conventions

- The examples presented here have been tested on `GNU bash` shell and **version 8.30** of the `GNU coreutils` package.
- Code snippets shown are copy pasted from `bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines have been added to improve readability, only `real` time is shown for speed comparisons and so on.
- Unless otherwise noted, all examples and explanations are meant for **ASCII** characters.
- External links are provided throughout the book for you to explore certain topics in more depth.
- The cli_text_processing_coreutils repo has all the code snippets, example files and other details related to the book. If you are not familiar with `git` command, click the **Code** button on the webpage to get the files.

Acknowledgements

- /r/commandline/, /r/linux4noobs/ and /r/linux/ — helpful forums
- stackoverflow and unix.stackexchange — for getting answers on pertinent questions related to cli tools
- tex.stackexchange — for help on pandoc and `tex` related questions
- canva — cover image
- Warning and Info icons by Amada44 under public domain
- pngquant and svgcleaner for optimizing images

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/cli_text_processing_coreutils/issues

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

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

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

License

Book version

1.0

See Version_changes.md to track changes across book versions.

Introduction

I've been using Linux since 2007, but it took me ten more years to *really* explore coreutils for my Command Line Text Processing repository.

Any beginner learning Linux command line tools would come across `cat` within the first week. Sooner or later, they'll come to know popular text processing tools like `grep` , `head` , `tail` , `tr` , `sort` , etc. If you were like me, you'd come across `sed` and `awk` , shudder at their complexity and prefer to use a scripting language like Perl and text editors like Vim instead (don't worry, I've already corrected that mistake).

Knowing power tools like `grep` , `sed` and `awk` can help solve most of your text processing needs. So, why would you want to learn text processing tools from the coreutils package? The biggest motivation would be faster execution since these tools are optimized for the use cases they solve. And there's always the advantage of not having to write code (and test that solution) if there's an existing tool to solve the problem.

This book will teach you more than twenty of such specialized text processing tools provided by the `GNU coreutils` package. Plenty of examples are provided to make it easier to understand a particular tool and its various features.

Writing a book always has a few pleasant surprises for me. For this one, it was discovering a `sort` option for calendar months, regular expression based features of `tac` and `nl` commands, etc.

Installation

On a GNU/Linux based OS, you are most likely to already have GNU coreutils installed. This book covers **version 8.30** of the coreutils package. To install a newer/particular version, see Coreutils download section for links and details.

If you are not using a Linux distribution, you may be able to access coreutils using these options:

- WSL
- brew

Documentation

It is always a good idea to know where to find the documentation. From the command line, you can use the `man` and `info` commands for brief manual and full documentation respectively. I prefer using the online GNU coreutils manual which feels much easier to use and navigate.

See also:

- Release notes — overview of changes and bug fixes between versions
- Bug list
- Common options
- FAQ
- Gotchas

cat and tac

`cat` derives its name from con**cat**enation and provides other nifty options too.

`tac` helps you to reverse the input line wise, usually used for further text processing.

Creating text files

Yeah, `cat` can be used to write contents to a file by typing them from the terminal itself. If you invoke `cat` without providing file arguments or `stdin` from a pipe, it will wait for you to type the content. After you are done typing all the text you want to save, press Enter key and then `Ctrl+d` key combination. If you don't want the last line to have a newline character, press `Ctrl+d` twice instead of Enter and `Ctrl+d`. See also unix.stackexchange: difference between Ctrl+c and Ctrl+d.

```
# press Enter key and Ctrl+d after typing all the required characters
$ cat > greeting.txt
Hi there
Have a nice day
```

In the above example, the output of `cat` is redirected to a file named `greeting.txt`. If you don't redirect the `stdout`, each line will be echoed as you type. You can check the contents of the file you just created by using `cat` again.

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

Here Documents is another popular way to create such files. Especially in shell scripts, since pressing `Ctrl+d` interactively won't be possible. Here's an example:

```
# > and a space at the start of lines are only present in interactive mode
# don't type them in a shell script
# EOF is typically used as the identifier
$ cat << 'EOF' > fruits.txt
> banana
> papaya
> mango
> EOF

$ cat fruits.txt
banana
papaya
mango
```

The termination string is enclosed in single quotes to prevent parameter expansion, command substitution, etc. You can also use `\string` for this purpose. If you use `<<-` instead of `<<`, you can use leading tab characters for indentation purposes. See bash manual: Here Documents and stackoverflow: here-documents for more details.

Note that creating files as shown above isn't restricted to `cat`, it can be applied to any command waiting for `stdin`.

```
# 'tr' converts lowercase alphabets to uppercase in this example
$ tr 'a-z' 'A-Z' << 'end' > op.txt
> hi there
> have a nice day
> end

$ cat op.txt
HI THERE
HAVE A NICE DAY
```

Concatenate files

Here's some examples to showcase `cat` 's main utility. One or more files can be given as arguments.

Visit the cli_text_processing_coreutils repo to get all the example files used in this book.

```
$ cat greeting.txt fruits.txt nums.txt
Hi there
Have a nice day
banana
papaya
mango
3.14
42
1000
```

To save the output of concatenation, use your shell's redirection features.

```
$ cat greeting.txt fruits.txt nums.txt > op.txt

$ cat op.txt
Hi there
Have a nice day
banana
papaya
mango
3.14
42
1000
```

Accepting stdin data

You can represent `stdin` data using `-` as a file argument. If file arguments are not present, `cat` will read from `stdin` data if present or wait for interactive input as seen earlier.

```
# only stdin (- is optional in this case)
$ echo 'apple banana cherry' | cat
apple banana cherry
```

```
# both stdin and file arguments
$ echo 'apple banana cherry' | cat greeting.txt -
Hi there
Have a nice day
apple banana cherry

# here's an example without newline character at the end of first input
$ printf 'Some\nNumbers' | cat - nums.txt
Some
Numbers3.14
42
1000
```

Squeeze consecutive empty lines

As mentioned before, `cat` provides many features beyond concatenation. Consider this sample `stdin` data:

```
$ printf 'hello\n\n\nworld\n\nhave a nice day\n'
hello


world

have a nice day
```

You can use the `-s` option to squeeze consecutive empty lines to a single empty line. If present, leading and trailing empty lines will also be squeezed, won't be completely removed. You can modify the below example to test it out.

```
$ printf 'hello\n\n\nworld\n\nhave a nice day\n' | cat -s
hello

world

have a nice day
```

Prefix line numbers

The `-n` option will prefix line number and a tab character to each input line. The line numbers are right justified to occupy a minimum of `6` characters, with space as the filler.

```
$ cat -n greeting.txt fruits.txt nums.txt
     1	Hi there
     2	Have a nice day
     3	banana
     4	papaya
     5	mango
     6	3.14
     7	42
     8	1000
```

Use `-b` option instead of `-n` option if you don't want empty lines to be numbered.

```
# -n option numbers all the input lines
$ printf 'apple\n\nbanana\n\ncherry\n' | cat -n
     1  apple
     2  
     3  banana
     4  
     5  cherry

# -b option numbers only the non-empty input lines
$ printf 'apple\n\nbanana\n\ncherry\n' | cat -b
     1  apple

     2  banana

     3  cherry
```

Use `nl` command if you want more customization options for numbering.

Viewing special characters

Characters like backspace and carriage return will mangle the contents if viewed naively on the terminal. Characters like NUL won't even be visible. You can use the `-v` option to show such characters using the caret notation (see wikipedia: Control code chart for details). See this unix.stackexchange thread for non-ASCII character examples.

```
# example for backspace and carriage return
$ printf 'car\bt\nbike\rp\n'
cat
pike
$ printf 'car\bt\nbike\rp\n' | cat -v
car^Ht
bike^Mp

# NUL character
$ printf 'car\0jeep\0bus\0' | cat -v
car^@jeep^@bus^@

# form-feed and vertical-tab
$ printf '1 2\t3\f4\v5\n' | cat -v
1 2     3^L4^K5
```

The `-v` option doesn't cover the newline and tab characters. You can use the `-T` option to spot tab characters.

```
$ printf 'good food\tnice dice\n' | cat -T
good food^Inice dice
```

The `-E` option adds a `$` marker at the end of input lines. This is useful to spot invisible trailing characters.

```
$ printf 'ice   \nwater\n cool  \n' | cat -E
ice   $
water$
 cool  $
```

The following options combine two or more of the above options:

- `-e` option is equivalent to `-vE`
- `-t` option is equivalent to `-vT`
- `-A` option is equivalent to `-vET`

```
$ printf 'car\bt\nbike\rp\n' | cat -e
car^Ht$
bike^Mp$

$ printf '1 2\t3\f4\v5\n' | cat -t
1 2^I3^L4^K5

$ printf '1 2\t3\f4\v5\n' | cat -A
1 2^I3^L4^K5$
```

Useless use of cat

Using `cat` to view the contents of a file, to concatenate them, etc is well and good. But, using `cat` when it is not needed is a bad habit that you should avoid. See wikipedia: UUOC and Useless Use of Cat Award for more details.

Most commands that you'll see in this book can directly work with file arguments, so you shouldn't use `cat` and pipe the contents for such cases. Here's a single file example:

```
# useless use of cat
$ cat greeting.txt | sed -E 's/\w+/\L\u&/g'
Hi There
Have A Nice Day

# sed can handle file arguments
$ sed -E 's/\w+/\L\u&/g' greeting.txt
Hi There
Have A Nice Day
```

If you prefer having the file argument before the command, you can still use your shell's redirection feature to supply input data instead of `cat` . This also applies to commands like `tr` that do not accept file arguments.

```
# useless use of cat
$ cat greeting.txt | tr 'a-z' 'A-Z'
HI THERE
HAVE A NICE DAY

# use shell redirection instead
$ <greeting.txt tr 'a-z' 'A-Z'
HI THERE
HAVE A NICE DAY
```

Such useless use of `cat` might not have a noticeable negative impact unless you are dealing with large input files. Especially for commands like `tac` and `tail` which will have to wait for all the data to be read instead of directly processing from the end of the file if they had been passed as arguments (or using shell redirection).

If you are dealing with multiple files, then the use of `cat` will depend upon the results desired. Here's some examples:

```
# match lines containing 'o' or '0'
# -n option adds line number prefix
$ cat greeting.txt fruits.txt nums.txt | grep -n '[o0]'
5:mango
8:1000
$ grep -n '[o0]' greeting.txt fruits.txt nums.txt
fruits.txt:3:mango
nums.txt:3:1000

# count the number of lines containing 'o' or '0'
$ grep -c '[o0]' greeting.txt fruits.txt nums.txt
greeting.txt:0
fruits.txt:1
nums.txt:1
$ cat greeting.txt fruits.txt nums.txt | grep -c '[o0]'
2
```

For some use cases like in-place editing with `sed`, you can't use `cat` or shell redirection at all. The files have to be passed as arguments only. To conclude, don't use `cat` just to pass the input as `stdin` for another command unless you really need to.

tac

`tac` will display the input lines in reversed order. If you pass multiple input files, each file content will be reversed separately. Here's some examples:

```
# won't be same as: cat greeting.txt fruits.txt | tac
$ tac greeting.txt fruits.txt
Have a nice day
Hi there
mango
papaya
banana

$ printf 'apple\nbanana\ncherry\n' | tac
cherry
banana
apple
```

⚠ If the last line of input doesn't end with a newline, the output will also not have that newline character.

```
$ printf 'apple\nbanana\ncherry' | tac
cherrybanana
apple
```

Reversing input lines makes some of the text processing tasks easier. For example, if there multiple matches but you want only the last such match. See my ebooks on GNU sed and GNU awk for more such use cases.

```
$ cat log.txt
--> warning 1
a,b,c,d
42
--> warning 2
x,y,z
--> warning 3
4,3,1

$ tac log.txt | grep -m1 'warning'
--> warning 3

$ tac log.txt | sed '/warning/q' | tac
--> warning 3
4,3,1
```

The `log.txt` input file has multiple lines containing `warning` . The task is to fetch lines based on the last match. Tools like `grep` and `sed` have features to easily match the first occurrence, so applying `tac` on the input helps to reverse the condition from last match to first match. Another benefit is that the first `tac` will stop reading input contents after the match is found in the above examples.

Use the `rev` command if you want each input line to be reversed character wise.

Customize line separator for tac

By default, the newline character is used to split the input content into *lines*. You can use the `-s` option to specify a different string to be used as the separator.

```
# use NUL as the line separator
# -s $'\0' can also be used instead of -s '' if ANSI-C quoting is supported
$ printf 'car\0jeep\0bus\0' | tac -s '' | cat -v
bus^@jeep^@car^@

# as seen before, last entry should also have the separator
# otherwise it won't be present in the output
$ printf 'apple banana cherry' | tac -s ' ' | cat -e
cherrybanana apple $
$ printf 'apple banana cherry ' | tac -s ' ' | cat -e
cherry banana apple $
```

When the custom separator occurs before the content of interest, use the `-b` option to print those separators before the content in the output as well.

```
$ cat body_sep.txt
%=%=
apple
banana
%=%=
red
green

$ tac -b -s '%=%=' body_sep.txt
%=%=
red
green
%=%=
apple
banana
```

The separator will be treated as a regular expression if you use the `-r` option as well.

```
$ cat shopping.txt
apple   50
toys    5
Pizza   2
mango   25
Banana  10

# separator character is 'a' or 'm' at the start of a line
$ tac -b -rs '^[am]' shopping.txt
mango   25
Banana  10
apple   50
toys    5
Pizza   2

# alternate solution for: tac log.txt | sed '/warning/q' | tac
# separator is zero or more characters from the start of a line till 'warning'
$ tac -b -rs '^.*warning' log.txt | awk '/warning/ && ++c==2{exit} 1'
--> warning 3
4,3,1
```

See Regular Expressions chapter from my **GNU grep** ebook if you want to learn about regexp syntax and features.

head and tail

`cat` is useful to view entire contents of file(s). Pagers like `less` can be used if you are working with large files (`man` pages for example). Sometimes though, you just want a peek at the starting or ending lines of input files. Or, you know the line numbers for the information you are looking for. In such cases, you can use `head` or `tail` or a combination of both these commands to extract the content you want.

Leading and trailing lines

Consider this sample file, with line numbers prefixed for convenience.

```
$ cat sample.txt
 1) Hello World
 2)
 3) Hi there
 4) How are you
 5)
 6) Just do-it
 7) Believe it
 8)
 9) banana
10) papaya
11) mango
12)
13) Much ado about nothing
14) He he he
15) Adios amigo
```

By default, `head` and `tail` will display the first and last 10 lines respectively.

```
$ head sample.txt
 1) Hello World
 2)
 3) Hi there
 4) How are you
 5)
 6) Just do-it
 7) Believe it
 8)
 9) banana
10) papaya

$ tail sample.txt
 6) Just do-it
 7) Believe it
 8)
 9) banana
10) papaya
11) mango
12)
13) Much ado about nothing
```

```
14) He he he
15) Adios amigo
```

If there are less than 10 lines in the input, only those lines will be displayed.

```
# seq command will be discussed in detail later, generates 1 to 4 here
# same as: seq 4 | tail
$ seq 4 | head
1
2
3
4
```

You can use the `-nN` option to customize the number of lines (`N`) needed.

```
# first three lines
# space between -n and N is optional
$ head -n3 sample.txt
 1) Hello World
 2) 
 3) Hi there

# last two lines
$ tail -n2 sample.txt
14) He he he
15) Adios amigo
```

Excluding the last N lines

By using `head -n -N` , you can get all the input lines except the ones you'll get when you use the `tail -nN` command.

```
# except the last 11 lines
# space between -n and -N is optional
$ head -n -11 sample.txt
 1) Hello World
 2) 
 3) Hi there
 4) How are you
```

Starting from Nth line

By using `tail -n +N` , you can get all the input lines except the ones you'll get when you use the `head -n(N-1)` command.

```
# all lines starting from the 11th line
# space between -n and +N is optional
$ tail -n +11 sample.txt
11) mango
12) 
13) Much ado about nothing
14) He he he
```

```
15) Adios amigo
```

Multiple input files

If you pass multiple input files to the `head` and `tail` commands, each file will be processed separately. By default, the output is nicely formatted with filename headers and empty line separators.

```
$ seq 2 | head -n1 greeting.txt -
==> greeting.txt <==
Hi there

==> standard input <==
1
```

You can use the `-q` option to avoid filename headers and empty line separators.

```
$ tail -q -n2 sample.txt nums.txt
14) He he he
15) Adios amigo
42
1000
```

Byte selection

The `-c` option works similar to the `-n` option, but with bytes instead of lines. In the below examples, newline characters have been added to the output for illustration purposes.

```
# first three characters
$ printf 'apple pie' | head -c3
app

# last three characters
$ printf 'apple pie' | tail -c3
pie

# excluding last four characters
$ printf 'car\njeep\nbus\n' | head -c -4
car
jeep

# all characters starting from fifth character
$ printf 'car\njeep\nbus\n' | tail -c +5
jeep
bus
```

Since `-c` works byte wise, it may not be suitable for multibyte characters:

```
# all input characters in this example occupy two bytes each
$ printf 'αλεπού' | head -c2
α

# ğ occupies three bytes
```

```
$ printf 'cağe' | tail -c4
ğe
```

Range of lines

You can select a range of lines by combining both `head` and `tail` commands.

```
# 9th to 11th lines
# same as: head -n11 sample.txt | tail -n3
$ tail -n +9 sample.txt | head -n3
 9) banana
10) papaya
11) mango

# 6th to 7th lines
# same as: tail -n +6 sample.txt | head -n2
$ head -n7 sample.txt | tail -n2
 6) Just do-it
 7) Believe it
```

See unix.stackexchange: line X to line Y on a huge file for performance comparison with other commands like `sed` , `awk` , etc.

NUL separator

The `-z` option sets the NUL character as the line separator instead of the newline character.

```
$ printf 'car\0jeep\0bus\0' | head -z -n2 | cat -v
car^@jeep^@

$ printf 'car\0jeep\0bus\0' | tail -z -n2 | cat -v
jeep^@bus^@
```

Further Reading

- wikipedia: File monitoring with tail -f and -F options
- unix.stackexchange: How does the tail -f option work?
- How to deal with output buffering?

tr

`tr` helps you to map one set of characters to another set of characters. Features like range, repeats, character sets, squeeze, complement, etc makes it a must know text processing tool.

To be precise, `tr` can handle only bytes. Multibyte character processing isn't supported yet.

Translation

Here's some examples that map one set of characters to another. As a good practice, always enclose the sets in single quotes to avoid issues due to shell metacharacters.

```
# 'l' maps to '1', 'e' to '3', 't' to '7' and 's' to '5'
$ echo 'leet speak' | tr 'lets' '1375'
1337 5p3ak

# example with shell metacharacters
$ echo 'apple;banana;cherry' | tr ; :
tr: missing operand
Try 'tr --help' for more information.
$ echo 'apple;banana;cherry' | tr ';' ':'
apple:banana:cherry
```

You can use `-` between two characters to construct a range (ascending order only).

```
# uppercase to lowercase
$ echo 'HELLO WORLD' | tr 'A-Z' 'a-z'
hello world

# swap case
$ echo 'Hello World' | tr 'a-zA-Z' 'A-Za-z'
hELLO wORLD

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

`tr` works only on `stdin` data, so use shell input redirection for file input.

```
$ tr 'a-z' 'A-Z' <greeting.txt
HI THERE
HAVE A NICE DAY
```

Different length sets

If the second set is longer, the extra characters are simply ignored. If the first set is longer, the last character of the second set is reused for the missing mappings.

```
# only abc gets converted to uppercase
$ echo 'apple banana cherry' | tr 'abc' 'A-Z'
Apple BAnAnA Cherry
```

```
# c-z will be converted to C
$ echo 'apple banana cherry' | tr 'a-z' 'ABC'
ACCCC BACACA CCCCCC
```

You can use the `-t` option to truncate the first set so that it matches the length of the second set.

```
# d-z won't be converted
$ echo 'apple banana cherry' | tr -t 'a-z' 'ABC'
Apple BAnAnA Cherry
```

You can also use `[c*n]` notation to repeat a character `c` by `n` times. You can specify `n` in decimal format or octal format (starts with `0`). If `n` is omitted, the character `c` is repeated as many times as needed to equalize the length of the sets.

```
# a-e will be translated to A
# f-z will be uppercased
$ echo 'apple banana cherry' | tr 'a-z' '[A*5]F-Z'
APPLA AANANA AHARRY

# a-c and x-z will be uppercased
# rest of the characters will be translated to -
$ echo 'apple banana cherry' | tr 'a-z' 'ABC[-*]XYZ'
A---- BA-A-A C----Y
```

Escape sequences and character sets

Certain characters like newline, tab, etc can be represented using escape sequences. You can also specify characters using `\NNN` octal representation.

```
# same as: tr '\011' '\072'
$ printf 'apple\tbanana\tcherry\n' | tr '\t' ':'
apple:banana:cherry

$ echo 'apple:banana:cherry' | tr ':' '\n'
apple
banana
cherry
```

Certain commonly useful groups of characters like alphabets, digits, punctuation, etc have named character sets that you can use instead of manually creating the sets. Only `[:lower:]` and `[:upper:]` can be used by default, others will require `-d` or `-s` options.

```
# same as: tr 'a-z' 'A-Z' <greeting.txt
$ tr '[:lower:]' '[:upper:]' <greeting.txt
HI THERE
HAVE A NICE DAY
```

To override the special meaning for `-` and `\` characters, you can escape them using the `\` character. You can also place the `-` character at the end of a set to represent it literally. Can you reason out why placing the `-` character at the start of a set can cause issues?

```
$ echo '/python-projects/programs' | tr '/-' '\\_'
\python_projects\programs
```

See tr manual for more details and a list of all the escape sequences and character sets.

Deleting characters

Use the `-d` option to specify a set of characters to be deleted.

```
$ echo '2021-08-12' | tr -d '-'
20210812

$ s='"Hi", there! How *are* you? All fine here.'
$ echo "$s" | tr -d '[:punct:]'
Hi there How are you All fine here
```

Complement

The `-c` option will invert the first set of characters. This is often used in combination with the `-d` option.

```
$ s='"Hi", there! How *are* you? All fine here.'

# retain alphabets, whitespaces, period, exclamation and question mark
$ echo "$s" | tr -cd 'a-zA-Z.!?[:space:]'
Hi there! How are you? All fine here.
```

If you use `-c` for translation, you can only provide a single character for the second set. In other words, all the characters except those provided by the first set will be mapped to the character specified by the second set.

```
$ s='"Hi", there! How *are* you? All fine here.'

$ echo "$s" | tr -c 'a-zA-Z.!?[:space:]' '1%'
tr: when translating with complemented character classes,
string2 must map all characters in the domain to one

$ echo "$s" | tr -c 'a-zA-Z.!?[:space:]' '%'
%Hi%% there! How %are% you? All fine here.
```

Squeeze

The `-s` option changes consecutive repeated characters to a single copy of that character.

```
# squeeze lowercase alphabets
$ echo 'hhoowwww aaaaaareeeeee yyouuuu!!' | tr -s 'a-z'
how are you!!

# translate and squeeze
$ echo 'hhoowwww aaaaaareeeeee yyouuuu!!' | tr -s 'a-z' 'A-Z'
HOW ARE YOU!!

# delete and squeeze
```

```
$ echo 'hhoowwww aaaaaareeeeee yyouuuu!!' | tr -sd '!' 'a-z'
how are you

# squeeze other than lowercase alphabets
$ echo 'how    are     you!!!!!' | tr -cs 'a-z'
how are you!
```

cut

`cut` is a handy tool for many field processing use cases. The features are limited compared to `awk` and `perl` commands, but the reduced scope also leads to faster processing.

Individual field selections

By default, `cut` splits the input content into fields based on the tab character. You can use the `-f` option to select a desired field from each input line. To extract multiple fields, specify the selections separated by the comma character.

```
# second field
$ printf 'apple\tbanana\tcherry\n' | cut -f2
banana

# first and third field
$ printf 'apple\tbanana\tcherry\n' | cut -f1,3
apple   cherry
```

`cut` will always display the selected fields in ascending order. Field duplication will be ignored as well.

```
# same as: cut -f1,3
$ printf 'apple\tbanana\tcherry\n' | cut -f3,1
apple   cherry

# same as: cut -f1,2
$ printf 'apple\tbanana\tcherry\n' | cut -f1,1,2,1,2,1,1,2
apple   banana
```

By default, `cut` uses the newline character as the line separator. `cut` will add a newline character to the output even if the last input line doesn't end with a newline.

```
$ printf 'good\tfood\ntip\ttap' | cut -f2
food
tap
```

Field ranges

You can use the `-` character to specify field ranges. You can skip the starting or ending range, but not both.

```
# 2nd, 3rd and 4th fields
$ printf 'apple\tbanana\tcherry\tdates\n' | cut -f2-4
banana  cherry  dates

# all fields from the start till the 3rd field
$ printf 'apple\tbanana\tcherry\tdates\n' | cut -f-3
apple   banana  cherry

# all fields from the 3rd field till the end
$ printf 'apple\tbanana\tcherry\tdates\n' | cut -f3-
cherry  dates
```

Input field delimiter

Use the `-d` option to change the input delimiter. Only a single byte character is allowed. By default, the output delimiter will be same as the input delimiter.

```
$ cat scores.csv
Name,Maths,Physics,Chemistry
Ith,100,100,100
Cy,97,98,95
Lin,78,83,80

$ cut -d, -f2,4 scores.csv
Maths,Chemistry
100,100
97,95
78,80

# use quotes if the delimiter is a shell metacharacter
$ echo 'one;two;three;four' | cut -d; -f3
cut: option requires an argument -- 'd'
Try 'cut --help' for more information.
-f3: command not found
$ echo 'one;two;three;four' | cut -d';' -f3
three
```

Output field delimiter

Use the `--output-delimiter` option to customize the output separator to any string of your choice. The string is treated literally. Depending on your shell you can use ANSI-C quoting to allow escape sequences.

```
# same as: tr '\t' ','
$ printf 'apple\tbanana\tcherry\n' | cut --output-delimiter=, -f1-
apple,banana,cherry

# multicharacter example
$ echo 'one;two;three;four' | cut -d';' --output-delimiter=' : ' -f1,3-
one : three : four

# ANSI-C quoting example
# depending on your environment, you can also press Ctrl+v and then Tab key
$ echo 'one;two;three;four' | cut -d';' --output-delimiter=$'\t' -f1,3-
one     three   four

# newline as the output field separator
$ echo 'one;two;three;four' | cut -d';' --output-delimiter=$'\n' -f2,4
two
four
```

Complement

The `--complement` option allows you to invert the field selections.

```
# except second field
$ printf 'apple ball cat\n1 2 3 4 5' | cut --complement -d' ' -f2
apple cat
1 3 4 5

# except first and third fields
$ printf 'apple ball cat\n1 2 3 4 5' | cut --complement -d' ' -f1,3
ball
2 4 5
```

Suppress lines without delimiters

By default, lines not containing the input delimiter will still be part of the output. You can use the `-s` option to suppress such lines.

```
$ cat mixed_fields.csv
1,2,3,4
hello
a,b,c

# second line doesn't have the comma separator
# by default, such lines will be part of the output
$ cut -d, -f2 mixed_fields.csv
2
hello
b

# use -s option to suppress such lines
$ cut -sd, -f2 mixed_fields.csv
2
b

$ cut --complement -sd, -f2 mixed_fields.csv
1,3,4
a,c
```

If a line contains the specified delimiter but doesn't have the field number requested, you'll get a blank line. The `-s` option has no effect on such lines.

```
$ printf 'apple ball cat\n1 2 3 4 5' | cut -d' ' -f4

4
```

Character selections

You can use the `-b` or `-c` options to select specified bytes from each input line. The syntax is same as the `-f` option. The `-c` option is intended for multibyte character selection, but for now it works exactly as the `-b` option. Character selection is useful for working with fixed-width fields.

```
$ printf 'apple\tbanana\tcherry\n' | cut -c2,8,11
pan
```

```
$ printf 'apple\tbanana\tcherry\n' | cut -c2,8,11 --output-delimiter=-
p-a-n

$ printf 'apple\tbanana\tcherry\n' | cut -c-5
apple

$ printf 'apple\tbanana\tcherry\n' | cut --complement -c13-
apple   banana

$ printf 'cat-bat\ndog:fog\nget;pet' | cut -c5-
bat
fog
pet
```

NUL separator

Use `-z` option if you want to use NUL character as the line separator. In this scenario, `cut` will ensure to add a final NUL character even if not present in the input.

```
$ printf 'good-food\0tip-tap\0' | cut -zd- -f2 | cat -v
food^@tap^@
```

Alternatives

Here's some alternate commands you can explore if `cut` isn't enough to solve your task.

- hck — supports regexp delimiters, field reordering, header based selection, etc
- xsv — fast CSV command line toolkit
- rcut — my `bash+awk` script, supports regexp delimiters, field reordering, negative indexing, etc
- awk — my ebook on `GNU awk` one-liners
- perl — my ebook on `perl` one-liners

seq

The `seq` command is a handy tool to generate a sequence of numbers in ascending or descending order. Both integer and floating-point numbers are supported. You can also customize the formatting for numbers and the separator between them.

Integer sequences

You need three numbers to generate an arithmetic progression — **start**, **step** and **stop**. When you pass only a single number as the stop value, the default start and step values are assumed to be `1`.

```
# start=1, step=1 and stop=3
$ seq 3
1
2
3
```

Passing two numbers are considered as start and stop values (in that order).

```
# start=25434, step=1 and stop=25437
$ seq 25434 25437
25434
25435
25436
25437

# start=-5, step=1 and stop=-3
$ seq -5 -3
-5
-4
-3
```

When you want to specify all the three numbers, the order is start, step and stop.

```
# start=1000, step=5 and stop=1010
$ seq 1000 5 1010
1000
1005
1010
```

By using a negative step value, you can generate sequences in descending order.

```
# no output
$ seq 3 1

# need to explicitly use a negative step value
$ seq 3 -1 1
3
2
1

$ seq 5 -5 -10
5
```

```
0
-5
-10
```

Floating-point sequences

Since `1` is the default start and step values, you need to change at least one of them to get floating-point sequences.

```
$ seq 0.5 3
0.5
1.5
2.5

$ seq 0.25 0.33 1.12
0.25
0.58
0.91
```

E scientific notation is also supported.

```
$ seq 1.2e2 1.22e2
120
121
122

$ seq 1.2e2 0.752 1.22e2
120.000
120.752
121.504
```

Customizing separator

You can use the `-s` option to change the separator between the numbers of a sequence. Multiple characters are allowed. Depending on your shell you can use ANSI-C quoting to use escapes like `\t` instead of a literal tab character. A newline is always added at the end of the output.

```
$ seq -s' ' 4
1 2 3 4

$ seq -s: -2 0.75 3
-2.00:-1.25:-0.50:0.25:1.00:1.75:2.50

$ seq -s' - ' 4
1 - 2 - 3 - 4

$ seq -s$'\n\n' 4
1

2

3
```

```
4
```

Leading zeros

By default, the output will not have leading zeros, even if they are part of the numbers passed to the command.

```
$ seq 008 010
8
9
10
```

The `-w` option will equalize the width of the output numbers using leading zeros. The largest width between the start and stop values will be used.

```
$ seq -w 8 10
08
09
10

$ seq -w 0003
0001
0002
0003
```

printf style formatting

You can use the `-f` option for `printf` style floating-point number formatting. See bash manual: printf for more details on formatting options.

```
$ seq -f'%g' -s: 1 0.75 3
1:1.75:2.5

$ seq -f'%.4f' -s: 1 0.75 3
1.0000:1.7500:2.5000

$ seq -f'%.3e' 1.2e2 0.752 1.22e2
1.200e+02
1.208e+02
1.215e+02
```

Limitations

As per the manual:

> On most systems, `seq` can produce whole-number output for values up to at least `2^53` . Larger integers are approximated. The details differ depending on your floating-point implementation.

```
# example with approximate values
$ seq 100000000000000000000 3 100000000000000000010
```

```
100000000000000000000
100000000000000000000
100000000000000000008
100000000000000000008
```

However, note that when limited to non-negative whole numbers, an increment of `1` and no format-specifying option, `seq` can print arbitrarily large numbers.

```
# no approximation for step value of 1
$ seq 100000000000000000000000000000 100000000000000000000000000005
100000000000000000000000000000
100000000000000000000000000001
100000000000000000000000000002
100000000000000000000000000003
100000000000000000000000000004
100000000000000000000000000005
```

shuf

The `shuf` command helps you randomize input lines. And there are features to limit the number of output lines, repeat lines and even generate random positive integers.

Randomize input lines

By default, `shuf` will randomize the order of input lines. Here's an example:

```
$ cat purchases.txt
coffee
tea
washing powder
coffee
toothpaste
tea
soap
tea

$ shuf purchases.txt
tea
coffee
tea
toothpaste
soap
coffee
washing powder
tea
```

You can use the `--random-source=FILE` option to provide your own source for randomness. With this option, the output will be the same across multiple runs. See Sources of random data for more details.

`shuf` doesn't accept multiple input files. Use `cat` for such cases.

Limit output lines

Use the `-n` option to limit the number of lines you want in the output. If the value is greater than the number of lines in the input, it would be similar to not using the `-n` option.

```
$ printf 'apple\nbanana\ncherry' | shuf -n2
cherry
apple
```

As seen in the example above, `shuf` will add a newline character if it is not present for the last input line.

Repeated lines

The `-r` option helps if you want to allow input lines to be repeated. This option is usually paired with `-n` to limit the number of lines in the output.

```
$ cat fruits.txt
banana
papaya
mango

$ shuf -n3 -r fruits.txt
banana
mango
banana

$ shuf -n5 -r fruits.txt
papaya
banana
mango
papaya
papaya
```

If a limit using `-n` is not specified, `shuf -r` will produce output lines indefinitely.

Specify input as arguments

You can use the `-e` option to specify multiple input lines as arguments to the command.

```
# quote the arguments as necessary
$ shuf -e hi there 'hello world' good
hello world
good
hi
there

$ shuf -n1 -e red green blue
blue

$ shuf -n4 -r -e red green blue
blue
green
red
blue
```

The shell will autocomplete unquoted glob patterns (provided there are files that match the given expression). You can thus easily construct a solution to get a random selection of files matching the given glob pattern.

```
$ echo *.csv
marks.csv mixed_fields.csv report_1.csv report_2.csv scores.csv
```

```
$ shuf -n2 -e *.csv
scores.csv
marks.csv
```

Generate random numbers

The `-i` option will help you generate random positive integers.

```
$ shuf -i 5-8
5
8
7
6

$ shuf -n3 -i 100-200
170
112
148

$ shuf -n5 -r -i 0-1
1
0
0
1
1
```

`2^64 - 1` is the maximum allowed integer when I tested it on my machine.

```
$ shuf -i 18446744073709551612-18446744073709551615
18446744073709551615
18446744073709551614
18446744073709551612
18446744073709551613

$ shuf -i 18446744073709551612-18446744073709551616
shuf: invalid input range: ‘18446744073709551616’:
Value too large for defined data type

# seq can help in such cases, but remember that shuf needs to read entire input
$ seq 100000000000000000000000000000 100000000000000000000000000105 | shuf -n2
100000000000000000000000000039
100000000000000000000000000018
```

`seq` can also help when you need negative and floating-point numbers.

```
$ seq -10 -8 | shuf
-9
-10
-8
```

```
$ seq -f'%.4f' 100 0.25 3000 | shuf -n3
1627.7500
1303.5000
2466.2500
```

See unix.stackexchange: generate random strings if numbers aren't enough for you.

Specifying output file

The `-o` option can be used to specify the output file to be used for saving the results. You can use this for in-place editing as well, since `shuf` reads the entire input before opening the output file.

```
$ shuf nums.txt -o rand_nums.txt

$ cat rand_nums.txt
42
1000
3.14
```

NUL separator

Use `-z` option if you want to use NUL character as the line separator. In this scenario, `shuf` will ensure to add a final NUL character even if not present in the input.

```
$ printf 'apple\0banana\0cherry' | shuf -z -n2 | cat -v
cherry^@banana^@
```