

CLI text processing with GNU Coreutils

```
#####
# NOTE: multibyte characters are not displayed correctly in pdf document. See what they really look like in #
#       txt document with the same name.                                     #
#####
```

1. Introduction
2. cat and tac
3. head and tail
4. tr
5. cut
6. seq
7. shuf
8. paste
9. pr
10. fold and fmt
11. sort
12. uniq
13. comm
14. join
15. nl
16. wc
17. split
18. csplit
19. expand and unexpand
20. basename and dirname

Introduction

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

Any beginner learning Linux command line tools would come across the cat command 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 and exercise 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 expressions in the 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 the version 9.1 of the coreutils package. To install a newer/particular version, see the coreutils download section for details.

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

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

cat and tac

cat derives its name from concatenation 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 data 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 and then the Ctrl+d key combinations. 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 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 data, 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. In this case, the termination condition is a line matching a predefined string which is specified after the `<<` redirection operator. This is especially helpful for automation, since pressing `Ctrl+d` interactively isn't desirable. Here's an example:

```
# > and a space at the start of lines represents the secondary prompt PS2
# 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 examples and details.

info 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 are some examples to showcase `cat`'s main utility. One or more files can be passed as arguments.

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

info Visit the [cli_text_processing_coreutils](#) repo to get all the example files used in this book.

To save the output of concatenation, use the 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 the stdin data using `-` as a file argument. If the file arguments are not present, `cat` will read the 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 a newline character at the end of the 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\n\n\n\napple\n'
hello
```

```
world
```

```
have a nice day
```

```
apple
```

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\n\n\n\napple\n' | cat -s
hello
```

```
world
```

```
have a nice day
```

```
apple
```

Prefix line numbers

The `-n` option will prefix line numbers 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 the `-b` option instead of `-n` 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 lines

```
$> printf 'apple\n\nbanana\n\ncherry\n' | cat -b
 1 apple
 2
 3 banana
 4
 5 cherry
```

info Use the `nl` command if you want more customization options like number formatting, separator string, regular expression based filtering and so on.

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](https://unix.stackexchange.com/questions/111111/showing-control-characters) thread for non-ASCII examples.

example for backspace and carriage return characters

```
$> printf 'mar\bt\nbike\rp\n'
```

```
mat
```

```
pike
```

```
$> printf 'mar\bt\nbike\rp\n' | cat -v
```

```
mar^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\napple\tbanana\tcherry\n' | cat -T
good food^Inice dice
apple^Ibanana^Icherry
```

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

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

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

```
$> printf 'mar\bt\nbike\rp\n' | cat -e
mar^Ht$
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 are 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 to 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 use the 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
$> &LTgreeting.txt tr 'a-z' 'A-Z'
HI THERE
HAVE A NICE DAY
```

Such useless use of cat might not have a noticeable negative impact for most cases. But it becomes important if 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 desired result. Here are 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 to another command, unless necessary.

tac
tac will reverse the order of the input lines. If you pass multiple input files, each file content will be reversed separately. Here are some examples:

```
# won't be the 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
```

warning If the last input line 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 are multiple matches but you want only the last one. 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
```

In the above example, log.txt has multiple lines containing warning. The task is to fetch lines based on the last match, which isn't usually supported by CLI tools. Matching the first occurrence is easy with tools like grep and sed. Hence, tac is helpful to reverse the condition from the last match to the first match. After processing with tools like sed, the result is then reversed again to get back the original order of input lines. Another benefit is that the first tac command will stop reading the input contents after the match is found.

info 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, the 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
%=%=
teal
green
```

```
$> tac -b -s '%=%=' body_sep.txt
%=%=
teal
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
```

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

Exercises

info All the exercises are also collated together in one place at `Exercises.md`. For solutions, see `Exercise_solutions.md`.

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

1) The given sample data has empty lines at the start and end of the input. Also, there are multiple empty lines between the paragraphs. How would you get the output shown below?

note that there's an empty line at the end of the output

```
$> printf '\n\nndragon\n\n\nunicorn\nbee\n\n\n' | ##### add your solution here
```

```
1 dragon
2 unicorn
3 bee
```

2) Pass appropriate arguments to the `cat` command to get the output shown below.

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

```
$> echo '42 apples and 100 bananas' | cat ##### add your solution here
```

```
42 apples and 100 bananas
Hi there
Have a nice day
```

3) What does the `-v` option of the `cat` command do?

4) Which options of the `cat` command do the following stand in for?

- * `-e` option is equivalent to
- * `-t` option is equivalent to
- * `-A` option is equivalent to

5) Will the two commands shown below produce the same output? If not, why not?

```
$> cat fruits.txt ip.txt | tac
```

```
$> tac fruits.txt ip.txt
```

6) Reverse the contents of `blocks.txt` file as shown below, considering `----` as the separator.

```
$> cat blocks.txt
----
apple--banana
mango---fig
----
3.14
-42
1000
----
sky blue
dark green
----
hi hello

##### add your solution here
----
hi hello
----
sky blue
dark green
----
3.14
-42
1000
----
apple--banana
mango---fig
```

7) For the blocks.txt file, write solutions to display only the last such group and last two groups.

```
##### add your solution here
----
hi hello

##### add your solution here
----
sky blue
dark green
----
hi hello
```

8) Reverse the contents of items.txt as shown below. Consider digits at the start of lines as the separator.

```
$> cat items.txt
1) fruits
apple 5
banana 10
2) colors
green
sky blue
3) magical beasts
dragon 3
unicorn 42

##### add your solution here
3) magical beasts
dragon 3
unicorn 42
2) colors
green
sky blue
1) fruits
apple 5
banana 10

---
head and tail
```

cat is useful to view entire contents of files. 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 3 here

same as: seq 3 | tail

```
$> seq 3 | head
```

```

1
2
3

```

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 the 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, the shell prompt at the end of the output aren't shown for illustration purposes.

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

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

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

# all characters starting from the 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 'İ±İ»İµİ\200İ¿İ\215' | head -c2
İ±

# gİ\210 requires three bytes
$> printf 'cagİ\210e' | tail -c4
gİ\210e
```

Range of lines

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

```
# 9th to 11th lines
# same as: head -n11 sample.txt | tail -n +9
$> 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 -n +6
6) Just do-it
7) Believe it
```

info See [unix.stackexchange: line X to line Y on a huge file for performance comparison with other commands like sed, awk, etc.](https://unix.stackexchange.com/questions/111111/head-and-tail-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^@
```

Exercises

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

1) Use appropriate commands and shell features to get the output shown below.

```
$> printf 'carpet\njeep\nbus\n'
carpet
jeep
bus
```

```
# use the above 'printf' command for input data
$> c=##### add your solution here
$> echo "$c"
car
```

2) How would you display all the input lines except the first one?

```
$> printf 'apple\nfig\ncarpet\njeep\nbus\n' | ##### add your solution here
fig
carpet
jeep
bus
```

3) Which command would you use to get the output shown below?

```
$> cat fruits.txt
banana
papaya
mango
$> cat blocks.txt
----
apple--banana
mango---fig
----
3.14
-42
1000
----
sky blue
dark green
----
hi hello

##### add your solution here
==> fruits.txt <==
banana
papaya

==> blocks.txt <==
----
apple--banana
```

4) Use a combination of head and tail commands to get the 11th to 14th characters from the given input.

```
$> printf 'apple\nfig\ncarpet\njeep\nbus\n' | ##### add your solution here
carp
```

5) Extract the starting six bytes from the input files ip.txt and fruits.txt.

```
##### add your solution here
it is banana
```

6) Extract the last six bytes from the input files fruits.txt and ip.txt.

```
##### add your solution here
mango
erish
```

7) For the input file ip.txt, display except the last 5 lines.

```
##### add your solution here
it is a warm and cozy day
listen to what I say
go play in the park
come back before the sky turns dark
```

8) Display the third line from the given stdin data. Consider the NUL character as the line separator.

```
$> printf 'apple\0fig\0carpet\0jeep\0bus\0' | ##### add your solution here
carpet

---
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.

Transliteration

Here are 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 inputs.

```
$> tr 'a-z' 'A-Z' &LTgreeting.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 the \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' &LTgreeting.txt
$> tr '[:lower:]' '[:upper:]' &LTgreeting.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
```

info See the 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 '2024-08-12' | tr -d '-'
20240812
```

```
# delete all punctuation characters
```

```
$> 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 transliteration, 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 'HELLO... hhoowwww aaaaaareeeeeee yyuuuuu!!' | tr -s 'a-z'
```

```
HELLO... how are you!!
```

```
# translate and squeeze
```

```
$> echo 'hhoowwww aaaaaareeeeeee yyuuuuu!!' | tr -s 'a-z' 'A-Z'
```

```
HOW ARE YOU!!
```

```
# delete and squeeze
```

```
$> echo 'hhoowwww aaaaaareeeeeee yyuuuuu!!' | tr -sd '!' 'a-z'
```

```
how are you
```

```
# squeeze other than lowercase alphabets
```

```
$> echo 'apple noon banana!!!!' | tr -cs 'a-z'
```

```
apple noon banana!
```

Exercises

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

1) What's wrong with the following command?

```
$> echo 'apple#banana#cherry' | tr # :
```

2) Retain only alphabets, digits and whitespace characters.

```
$> printf 'Apple_42 cool,blue\tDragon:army\n' | ##### add your solution here
```

```
Apple42 coolblue Dragonarmy
```

3) Similar to rot13, figure out a way to shift digits such that the same logic can be used both ways.

```
$> echo '4780 89073' | ##### add your solution here
```

```
9235 34528
```

```
$> echo '9235 34528' | ##### add your solution here
```

```
4780 89073
```

4) Figure out the logic based on the given input and output data. Hint: use two ranges for the first set and only 6 characters in the second set.

```
$> echo 'apple banana cherry damson etrog' | ##### add your solution here
```

```
1XX15 21n1n1 3h5XXX 41mXon 5XXog
```

5) Which option would you use to truncate the first set so that it matches the length of the second set?

6) What does the * notation do in the second set?

7) Change : to - and ; to the newline character.

```
$> echo 'tea:coffee;brown:teal;dragon:unicorn' | ##### add your solution here
tea-coffee
brown-teal
dragon-unicorn
```

8) Convert all characters to * except digit and newline characters.

```
$> echo 'ajsd45_sdg2Khnf4v_54as' | ##### add your solution here
****45****2****4**54**
```

9) Change consecutive repeated punctuation characters to a single punctuation character.

```
$> echo '"hi..."", good morning!!!!' | ##### add your solution here
"hi.", good morning!
```

10) Figure out the logic based on the given input and output data.

```
$> echo 'Aapple    noon    banana!!!!' | ##### add your solution here
:apple:noon:banana:
```

11) The books.txt file has items separated by one or more : characters. Change this separator to a single newline character as shown below.

```
$> cat books.txt
Cradle::Mage Errant::The Weirkey Chronicles
Mother of Learning::Eight::::Dear Spellbook:Ascendant
Mark of the Fool:Super Powereds::Ends of Magic
```

add your solution here

```
Cradle
Mage Errant
The Weirkey Chronicles
Mother of Learning
Eight
Dear Spellbook
Ascendant
Mark of the Fool
Super Powereds
Ends of Magic
```

```
---
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.

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

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

cut will always display the selected fields in ascending order. And you cannot display a field more than once.

```
# 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\tfig\tmango\n' | cut -f2-4
banana  cherry  fig
```

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

```
# all fields from the 3rd one till the end
$> printf 'apple\tbanana\tcherry\tfig\tmango\n' | cut -f3-
cherry fig      mango
```

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
```

```
# example for multicharacter output separator
$> 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 the 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 the second field
$> printf 'apple ball cat\n1 2 3 4 5' | cut --complement -d' ' -f2
apple cat
1 3 4 5
```

```
# except the 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 the -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
```

info 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 specific 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 the -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 are 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
- * choose - negative indexing, regexp based delimiters, 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

Exercises

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

1) Display only the third field.

```
$> printf 'tea\tcoffee\tchocolate\tfruit\n' | ##### add your solution here
chocolate
```

2) Display the second and fifth fields. Consider , as the field separator.

```
$> echo 'tea,coffee,chocolate,ice cream,fruit' | ##### add your solution here
coffee,fruit
```

3) Why does the below command not work as expected? What other tools can you use in such cases?

```
# not working as expected
$> echo 'apple,banana,cherry,fig' | cut -d, -f3,1,3
apple,cherry
```

```
# expected output
$> echo 'apple,banana,cherry,fig' | ##### add your solution here
cherry,apple,cherry
```

4) Display except the second field in the format shown below. Can you construct two different solutions?

```
# solution 1
$> echo 'apple,banana,cherry,fig' | ##### add your solution here
```

apple cherry fig

```
# solution 2
$> echo '2,3,4,5,6,7,8' | ##### add your solution here
2 4 5 6 7 8
```

5) Extract the first three characters from the input lines as shown below. Can you also use the head command for this purpose? If not, why not?

```
$> printf 'apple\nbanana\ncherry\nfig\n' | ##### add your solution here
app
ban
che
fig
```

6) Display only the first and third fields of the scores.csv input file, with tab as the output field separator.

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

```
##### add your solution here
Name      Physics
Ith       100
Cy        98
Lin       83
```

7) The given input data uses one or more : characters as the field separator. Assume that no field content will have the : character. Display except the second field, with : as the output field separator.

```
$> cat books.txt
Cradle:::Mage Errant::The Weirkey Chronicles
Mother of Learning::Eight::::Dear Spellbook:Ascendant
Mark of the Fool:Super Powered:::Ends of Magic
```

```
##### add your solution here
Cradle : The Weirkey Chronicles
Mother of Learning : Dear Spellbook : Ascendant
Mark of the Fool : Ends of Magic
```

8) Which option would you use to not display lines that do not contain the input delimiter character?

9) Modify the command to get the expected output shown below.

```
$> printf 'apple\nbanana\ncherry\n' | cut -c-3 --output-delimiter=:
app
ban
che
```

```
$> printf 'apple\nbanana\ncherry\n' | ##### add your solution here
a:p:p
b:a:n
c:h:e
```

10) Figure out the logic based on the given input and output data.

```
$> printf 'apple\0fig\0carpet\0jeep\0' | ##### add your solution here | cat -v
ple^@g^@rpet^@ep^@
```

```
---
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' 3
1
2
3
```

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
```


4) Is the sequence shown below possible to generate with seq? If so, how?

```
##### add your solution here
01.5,02.5,03.5,04.5,05.5
```

5) Modify the command shown below to customize the output numbering format.

```
$> seq 30.14 3.36 40.72
30.14
33.50
36.86
40.22
```

```
##### add your solution here
3.014e+01
3.350e+01
3.686e+01
4.022e+01
```

```
---
shuf
```

The shuf command helps you randomize the 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
```

info 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.

warning 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\nfig\nmango' | shuf -n2
mango
cherry
```

info As seen in the example above, shuf will add a newline character even 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
```

info 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. This is more useful for in-place editing, since you can simply use shell redirection to save the output in a different file.

```
$> cat book_list.txt
Cradle
Mage Errant
Mother of Learning
Super Powereds
The Umbral Storm
The Weirkey Chronicles

$> shuf book_list.txt -o book_list.txt
$> cat book_list.txt
Super Powereds
Cradle
Mage Errant
The Weirkey Chronicles
Mother of Learning
The Umbral Storm
```

NUL separator

Use the `-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\0fig\0mango' | shuf -z -n3 | cat -v
banana^@mango^@cherry^@
```

Exercises

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

1) What's wrong with the given command?

```
$> shuf --random-source=greeting.txt fruits.txt books.txt
shuf: extra operand 'books.txt'
Try 'shuf --help' for more information.
```

```
# expected output
##### add your solution here
banana
Cradle:::Mage Errant::The Weirkey Chronicles
Mother of Learning::Eight::::Dear Spellbook:Ascendant
papaya
Mark of the Fool:Super Powereds:::Ends of Magic
mango
```

2) What do the `-r` and `-n` options do? Why are they often used together?

3) What does the following command do?

```
$> shuf -e apple banana cherry fig mango
```

4) Which option would you use to generate random numbers? Given an example.

5) How would you generate 5 random numbers between 0.125 and 0.789 with a step value of 0.023?

```
# output shown below is a sample, might differ for you
##### add your solution here
0.378
0.631
0.447
0.746
0.723
```

```
---
paste
```

`paste` is typically used to merge two or more files column wise. It also has a handy feature for serializing data.

Concatenating files column wise

Consider these two input files:

```
$> cat colors_1.txt
Blue
Brown
Orange
Purple
Red
Teal
White

$> cat colors_2.txt
Black
```

```
Blue
Green
Orange
Pink
Red
White
```

By default, paste adds a tab character between corresponding lines of the input files.

```
$> paste colors_1.txt colors_2.txt
```

```
Blue    Black
Brown   Blue
Orange  Green
Purple  Orange
Red     Pink
Teal    Red
White   White
```

You can use the -d option to change the delimiter between the columns. The separator is added even if the data has been exhausted for some of the input files. Here are some examples with single character delimiters. Multicharacter separation will be discussed later.

```
$> seq 4 | paste -d, - <(seq 6 9)
```

```
1,6
2,7
3,8
4,9
```

```
# quote the delimiter if it is a shell metacharacter
```

```
$> paste -d'|' <(seq 3) <(seq 4 5) <(seq 6 8)
```

```
1|4|6
2|5|7
3||8
```

Use an empty string if you don't want any delimiter between the columns. You can also use \0 for this case, but that'd be confusing since it is typically used to mean the NUL character.

```
# note that the space between -d and empty string is necessary here
```

```
$> paste -d ' ' <(seq 3) <(seq 6 8)
```

```
16
27
38
```

info You can pass the same filename multiple times too - they will be treated as if they are separate inputs. This doesn't apply for stdin data though, which is a special case as discussed in a later section.

Interleaving lines

By setting the newline character as the delimiter, you'll get interleaved lines.

```
$> paste -d'\n' <(seq 11 13) <(seq 101 103)
```

```
11
101
12
102
13
103
```

Multiple columns from single input

If you use - multiple times, paste will consume a line from stdin data every time - is encountered. This is different from using the same filename multiple times, in which case they are treated as separate inputs.

This special case for stdin data is useful to combine consecutive lines using the given delimiter. Here are some examples to help you understand this feature better:

```
# two columns
```

```
$> seq 10 | paste -d, - -
```

```
1,2
3,4
5,6
7,8
9,10
```

```
# five columns
```

```
$> seq 10 | paste -d: - - - - -
```

```
1:2:3:4:5
6:7:8:9:10
```

```
# use shell redirection for file input
```

```
$> &LTcolors_1.txt paste -d: - - -
```

```
Blue:Brown:Orange
Purple:Red:Teal
White::
```

Here's an example with both stdin and file arguments:

```
$> seq 6 | paste - nums.txt -
```

```
1      3.14    2
3      42      4
5      1000    6
```

If you don't want to manually type the number of - required, you can use this printf trick:

```
# the string before %.s is repeated based on the number of arguments
$> printf 'x %.s' a b c
x x x
$> printf -- '- %.s' {1..5}
- - - - -
```

```
$> seq 10 | paste -d, $(printf -- '- %.s' {1..5})
1,2,3,4,5
6,7,8,9,10
```

info See this [stackoverflow thread](#) for more details about the printf solution and other alternatives.

Multicharacter delimiters

The -d option accepts a list of characters (bytes to be precise) to be used one by one between the different columns. If the number of characters is less than the number of separators required, the characters are reused from the beginning and this cycle repeats until all the columns are done. If the number of characters is greater than the number of separators required, the extra characters are simply discarded.

```
# , is used between the 1st and 2nd columns
# - is used between the 2nd and 3rd columns
$> paste -d',-' <(seq 3) <(seq 4 6) <(seq 7 9)
1,4-7
2,5-8
3,6-9

# only 3 separators are needed, the rest are discarded
$> paste -d',-:;.[\]' <(seq 3) <(seq 4 6) <(seq 7 9) <(seq 10 12)
1,4-7:10
2,5-8:11
3,6-9:12

# 2 characters given, 4 separators needed
# paste will reuse from the start of the list
$> seq 10 | paste -d':,' - - - -
1:2,3:4,5
6:7,8:9,10
```

You can use empty files to get multicharacter separation between the columns.

```
$> paste -d' : ' <(seq 3) /dev/null /dev/null <(seq 4 6)
1 : 4
2 : 5
3 : 6

# create an empty file to avoid typing /dev/null too many times
$> > e
$> paste -d' : - ' <(seq 3) e e <(seq 4 6) e e <(seq 7 9)
1 : 4 - 7
2 : 5 - 8
3 : 6 - 9
```

Serialize

The -s option allows you to combine all the input lines from a file into a single line using the given delimiter. paste will ensure to add a final newline character even if it wasn't present in the input.

```
# this will give you a trailing comma
# and there won't be a newline character at the end
$> &LTcolors_1.txt tr '\n' ','
Blue,Brown,Orange,Purple,Red,Teal,White,

# paste changes the separator between the lines only
# and there will be a newline character at the end
$> paste -sd, colors_1.txt
Blue,Brown,Orange,Purple,Red,Teal,White

# newline gets added at the end even if not present in the input
$> printf 'apple\nbanana\ncherry' | paste -sd-
apple-banana-cherry
```

If multiple files are passed, serialization of each file is displayed on separate lines.

```
$> paste -sd: colors_1.txt colors_2.txt
Blue:Brown:Orange:Purple:Red:Teal:White
Black:Blue:Green:Orange:Pink:Red:White
```

```
$> paste -sd, <(seq 3) <(seq 5 9)
1,2,3
5,6,7,8,9
```

NUL separator

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

```
$> printf 'a\0b\0c\0d\0e\0f\0g\0h' | paste -z -d: - - - | cat -v
a:b:c:d^@e:f:g:h^@
```

Exercises

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

1) What's the default delimiter character added by the `paste` command? Which option would you use to customize this separator?

2) Will the following two commands produce equivalent output? If not, why not?

```
$> paste -d, <(seq 3) <(printf '%s\n' item_{1..3})
```

```
$> printf '%s\n' {1..3},item_{1..3}
```

3) Combine the two data sources as shown below.

```
$> printf '1)\n2)\n3)'
```

```
1)
```

```
2)
```

```
3)
```

```
$> cat fruits.txt
```

```
banana
```

```
papaya
```

```
mango
```

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

```
1)banana
```

```
2)papaya
```

```
3)mango
```

4) Interleave the contents of `fruits.txt` and `books.txt`.

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

```
banana
```

```
Cradle:::Mage Errant::The Weirkey Chronicles
```

```
papaya
```

```
Mother of Learning::Eight::::Dear Spellbook:Ascendant
```

```
mango
```

```
Mark of the Fool:Super Powered:::Ends of Magic
```

5) Generate numbers 1 to 9 in two different formats as shown below.

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

```
1:2:3
```

```
4:5:6
```

```
7:8:9
```

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

```
1 : 4 : 7
```

```
2 : 5 : 8
```

```
3 : 6 : 9
```

6) Combine the contents of `fruits.txt` and `colors.txt` as shown below.

```
$> cat fruits.txt
```

```
banana
```

```
papaya
```

```
mango
```

```
$> cat colors.txt
```

```
deep blue
```

```
light orange
```

```
blue delight
```

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

```
banana,deep blue,papaya,light orange,mango,blue delight
```

```
---
```

```
pr
```

Paginate or columnate `FILE(s)` for printing.

As stated in the above quote from the manual, the `pr` command is mainly used for those two tasks. This book will discuss only the columnate features and some miscellaneous tasks.

Here's a pagination example if you are interested in exploring further. The `pr` command will add blank lines, a header and so on to make it suitable for printing.

```
$> pr greeting.txt | head
```



```
Hi there
Have a nice day
```

Columnate

The `--columns` and `-a` options can be used to merge the input lines in two different ways:

- * split the input file and then merge them as columns
- * merge consecutive lines, similar to the `paste` command

Here's an example to get started. Note that `-N` is same as using `--columns=N` where `N` is the number of columns you want in the output. The default page width is 72, which means each column can only have a maximum of $72/N$ characters (including the separator). Tab and space characters will be used to fill the columns as needed. You can use the `-J` option to prevent `pr` from truncating longer columns. The `-t` option is used here to turn off the pagination features.

```
# split input into three parts
# each column width is 72/3 = 24 characters max
$> seq 9 | pr -3t
1          4          7
2          5          8
3          6          9
```

You can customize the separator using the `-s` option. The default is a tab character which you can change to any other string value. The `-s` option also turns off line truncation, so the `-J` option isn't needed. However, the default page width of 72 can still cause issues, which will be discussed later.

```
# tab separator
$> seq 9 | pr -3ts
1      4      7
2      5      8
3      6      9
```

```
# comma separator
$> seq 9 | pr -3ts,
1,4,7
2,5,8
3,6,9
```

```
# multicharacter separator
$> seq 9 | pr -3ts' : '
1 : 4 : 7
2 : 5 : 8
3 : 6 : 9
```

Use the `-a` option to merge consecutive lines, similar to the `paste` command. One advantage is that the `-s` option supports a string value, whereas with `paste` you'd need to use workarounds to get multicharacter separation.

```
# four consecutive lines are merged
# same as: paste -d: - - - -
$> seq 8 | pr -4ats:
1:2:3:4
5:6:7:8
```

There are other differences between the `pr` and `paste` commands as well. Unlike `paste`, the `pr` command doesn't add the separator if the last row doesn't have enough columns. Another difference is that `pr` doesn't support an option to use the NUL character as the line separator.

```
$> seq 10 | pr -4ats,
1,2,3,4
5,6,7,8
9,10
```

```
$> seq 10 | paste -d, - - - -
1,2,3,4
5,6,7,8
9,10,,
```

Customizing page width

As mentioned before, the default page width is 72. This can cause lines to be truncated, unless the `-s` or `-J` options are used. There's another issue you might run into, for example:

```
$> seq 100 | pr -50ats,
pr: page width too narrow
```

$(N-1)*\text{length}(\text{separator}) + N$ is the minimum page width you need, where `N` is the number of columns required. So, for 50 columns and a separator of length 1, you'll need a minimum width of 99. This calculation doesn't make any assumption about the size of input lines, so you may need `-J` to ensure input lines aren't truncated.

You can use the `-w` option to change the page width. The `-w` option overrides the effect of `-s` option on line truncation, so use `-J` option as well unless you really need truncation. If truncation is active, maximum

column width is $(\text{PageWidth} - (N-1) * \text{length}(\text{separator})) / N$ rounded down to an integer value. Here are some examples:

```
# minimum width needed is 3 for N=2 and length=1
# maximum column width: (6 - 1) / 2 = 2
$> pr -w6 -2ts, greeting.txt
Hi,Ha
# use -J to avoid truncation
$> pr -J -w6 -2ts, greeting.txt
Hi there,Have a nice day

# N=3 and length=4, so minimum width needed is (3-1)*4 + 3 = 11
$> seq 6 | pr -J -w10 -3ats'::::'
pr: page width too narrow
$> seq 6 | pr -J -w11 -3ats'::::'
1::::2::::3
4::::5::::6

# you can also just use a large number to avoid having to calculate the width
$> seq 6 | pr -J -w500 -3ats'::::'
1::::2::::3
4::::5::::6
```

Concatenating files column wise

Two or more input files can be merged column wise using the `-m` option. As seen before, `-t` is needed to ignore pagination features and `-s` can be used to customize the separator.

```
# same as: paste colors_1.txt colors_2.txt
$> pr -mts colors_1.txt colors_2.txt
Blue    Black
Brown   Blue
Orange  Green
Purple  Orange
Red      Pink
Teal     Red
White    White

# same as: paste -d' : ' <(seq 3) /dev/null /dev/null <(seq 4 6)
$> pr -mts' : ' <(seq 3) <(seq 4 6)
1 : 4
2 : 5
3 : 6
```

You can prefix the output with line numbers using the `-n` option. By default, this option supports up to 5 digit numbers and uses the tab character to separate the numbering and line contents. You can optionally pass two arguments to this option - maximum number of digits and the separator character. If both arguments are used, the separator should be specified first. If you want to customize the starting line number, use the `-N` option as well.

```
# maximum of 1 digit for numbering
# use : as the separator between the line number and line contents
$> pr -n:1 -mts, colors_1.txt colors_2.txt
1:Blue,Black
2:Brown,Blue
3:Orange,Green
4:Purple,Orange
5:Red,Pink
6:Teal,Red
7:White,White
```

The string passed to `-s` is treated literally. Depending on your shell you can use ANSI-C quoting to allow escape sequences. Unlike `columnate`, the separator is added even if the data is missing for some of the files.

```
# greeting.txt has 2 lines
# fruits.txt has 3 lines
# same as: paste -d$'\n' greeting.txt fruits.txt
$> pr -mts$'\n' greeting.txt fruits.txt
Hi there
banana
Have a nice day
papaya
```

mango

Miscellaneous

You can use the `-d` option to double space the input contents. That is, every newline character is doubled.

```
$> pr -dt fruits.txt
banana

papaya

mango
```

The `-v` option will convert non-printing characters like carriage return, backspace, etc to their octal

representations (\NNN).

```
$> printf 'car\bt\r\nbike\0p\r\n' | pr -vt
car\010t\015
bike\000p\015
```

`pr -t` is a roundabout way of concatenating input files. But one advantage is that this will add a newline character at the end if not present in the input.

```
# 'cat' will not add a newline character
# so, use 'pr' if newline is needed at the end
$> printf 'a\nb\nc' | pr -t
a
b
c
```

Exercises

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

1) What does the `-t` option do?

2) Generate numbers 1 to 16 in two different formats as shown below.

```
$> seq -w 16 | ##### add your solution here
01,02,03,04
05,06,07,08
09,10,11,12
13,14,15,16
```

```
$> seq -w 16 | ##### add your solution here
01,05,09,13
02,06,10,14
03,07,11,15
04,08,12,16
```

3) How'd you solve the issue shown below?

```
$> seq 100 | pr -37ats,
pr: page width too narrow
```

4) Combine the contents of `fruits.txt` and `colors.txt` in two different formats as shown below.

```
$> cat fruits.txt
banana
papaya
mango
$> cat colors.txt
deep blue
light orange
blue delight
```

```
##### add your solution here
banana : deep blue
papaya : light orange
mango : blue delight
```

```
##### add your solution here
1:banana,deep blue
2:papaya,light orange
3:mango,blue delight
```

5) What does the `-d` option do?

```
---
fold and fmt
```

These two commands are useful to split and join lines to meet a specific line length requirement. `fmt` is smarter and usually the tool you want, but `fold` can be handy for some cases.

`fold`
By default, `fold` will wrap lines that are greater than 80 bytes long, which can be customized using the `-w` option. The newline character isn't part of this line length calculation. You might wonder if there are tasks where wrapping without context could be useful. One use case I can think of is the FASTA format.

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

```
# splits the second line since it is greater than 10 bytes
$> fold -w10 greeting.txt
Hi there
Have a nic
e day
```

The `-s` option looks for the presence of spaces to determine the line splitting. This check is performed

within the limits of the wrap length.

```
$> fold -s -w10 greeting.txt
Hi there
Have a
nice day
```

However, the `-s` option can still split words if there's no blank space before the specified width. Use `fmt` if you don't want this behavior.

```
$> echo 'hi there' | fold -s -w4
hi
ther
e
```

The `-b` option will cause `fold` to treat tab, backspace, and carriage return characters as if they were a single byte character.

```
# tab can occupy up to 8 columns
$> printf 'a\tb\tc\t1\t2\t3\n' | fold -w6
a
```

```
b
```

```
c
```

```
1
```

```
2
```

```
3
```

```
# here, tab will be treated as if it occupies only a single column
```

```
$> printf 'a\tb\tc\t1\t2\t3\n' | fold -b -w6
a      b      c
1      2      3
```

```
fmt
```

The `fmt` command makes a smarter decision based on sentences, paragraphs and other details. Here's an example that splits a single line (taken from the documentation of `fmt` command) into several lines. The default formatting is 93% of 75 columns. The `-w` option controls the width parameter and the `-g` option controls the percentage of columns.

```
$> fmt info_fmt.txt
fmt prefers breaking lines at the end of a sentence, and tries to
avoid line breaks after the first word of a sentence or before the last
word of a sentence. A sentence break is defined as either the end of a
paragraph or a word ending in any of '.?!', followed by two spaces or
end of line, ignoring any intervening parentheses or quotes. Like TeX,
fmt reads entire "paragraphs" before choosing line breaks; the algorithm
is a variant of that given by Donald E. Knuth and Michael F. Plass in
"Breaking Paragraphs Into Lines", Software-Practice & Experience 11,
11 (November 1981), 1119-1184.
```

Unlike the `fold` command, words are not split even if they exceed the maximum line width. Another difference is that `fmt` will add a final newline character even if it wasn't present in the input.

```
$> printf 'hi there' | fmt -w4
hi
there
```

The `fmt` command also allows you to join lines together that are shorter than the specified width. As mentioned before, paragraphs are taken into consideration, so empty lines will prevent merging. The `-s` option will disable line merging.

```
$> 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
```

```
# 'cut' here helps to ignore the first 4 characters of sample.txt
```

```
$> cut -c5- sample.txt | fmt -w30
Hello World
```

Hi there How are you

Just do-it Believe it

banana papaya mango

Much ado about nothing He he
he Adios amigo

The -u option will change multiple spaces to a single space. Excess spacing between sentences will be changed to two spaces.

```
$> printf 'Hi   there.   Have a nice   day\n' | fmt -u
Hi there.  Have a nice day
```

There are options that control indentation, formatting only lines with a specific prefix and so on. See `fmt` documentation for more details.

Exercises

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

1) What's the default wrap length of the `fold` and `fmt` commands?

2) Fold the given stdin data at 9 bytes.

```
$> echo 'hi hello, how are you?' | ##### add your solution here
hi hello,
  how are
you?
```

3) Figure out the logic based on the given input and output data using the `fold` command.

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

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

```
##### add your solution here
it is a
warm and
cozy day
listen to
what I say
```

4) What does the `fold -b` option do?

5) How'd you get the expected output shown below?

```
# wrong output
$> echo 'fig appleseed mango pomegranate' | fold -sw7
fig
applese
ed
mango
pomegra
nate
```

```
# expected output
$> echo 'fig appleseed mango pomegranate' | ##### add your solution here
fig
appleseed
mango
pomegranate
```

6) What do the options `-s` and `-u` of the `fmt` command do?

```
---
sort
```

The `sort` command provides a wide variety of features. In addition to lexicographic ordering, it supports various numerical formats. You can also sort based on particular columns. And there are nifty features like merging already sorted input, debugging, determining whether the input is already sorted and so on.

Default sort and Collating order

By default, `sort` orders the input in ascending order. If you know about ASCII codepoints, do you agree that the following two examples are showing the correct expected output?

```
$> cat greeting.txt
```

```

Hi there
Have a nice day
# extract and sort space separated words
$> &LTgreeting.txt tr ' ' '\n' | sort
a
day
Have
Hi
nice
there

```

```

$> printf '(banana)\n{cherry}\n[apple]' | sort
[apple]
(banana)
{cherry}

```

From the sort manual:

Unless otherwise specified, all comparisons use the character collating sequence specified by the LC_COLLATE locale.

If you use a non-POSIX locale (e.g., by setting LC_ALL to en_US), then sort may produce output that is sorted differently than you're accustomed to. In that case, set the LC_ALL environment variable to C. Note that setting only LC_COLLATE has two problems. First, it is ineffective if LC_ALL is also set. Second, it has undefined behavior if LC_CTYPE (or LANG, if LC_CTYPE is unset) is set to an incompatible value. For example, you get undefined behavior if LC_CTYPE is ja_JP.PCK but LC_COLLATE is en_US.UTF-8.

My locale settings are based on en_IN, which is different from the POSIX sorting order. So, the fact to remember is that sort obeys the rules of the current locale. If you want POSIX sorting, one option is to use LC_ALL=C as shown below.

```

$> &LTgreeting.txt tr ' ' '\n' | LC_ALL=C sort
Have
Hi
a
day
nice
there

```

```

$> printf '(banana)\n{cherry}\n[apple]' | LC_ALL=C sort
(banana)
[apple]
{cherry}

```

info Another benefit of C locale is that it will be significantly faster compared to Unicode parsing and sorting rules.

info Use the -f option if you want to explicitly ignore case. See also GNU Core Utilities FAQ: Sort does not sort in normal order!.

info See this [unix.stackexchange](#) thread if you want to create your own custom sort order.

Ignoring headers

You can use sed -u to consume only the header lines and leave the rest of the input for the sort command. Note that this unbuffered option is supported by GNU sed, might not be available with other implementations.

```

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

```

1q is used to quit after the first line

```

$> ( sed -u '1q' ; sort ) &LTscores.csv
Name,Maths,Physics,Chemistry
Cy,97,98,95
Ith,100,100,100
Lin,78,83,80

```

info See this [unix.stackexchange](#) thread for more ways of ignoring headers. See bash manual: Grouping Commands for more details about the () grouping used in the above example.

Dictionary sort

The -d option will consider only alphabets, numbers and blanks for sorting. Space and tab characters are considered as blanks, but this would also depend on the locale.

```

$> printf '(banana)\n{cherry}\n[apple]' | LC_ALL=C sort -d
[apple]
(banana)
{cherry}

```

info Use the -i option if you want to ignore only the non-printing characters.

Reversed order

The -r option will reverse the output order. Note that this doesn't change how sort performs comparisons, only the output is reversed. You'll see an example later where this distinction becomes clearer.

```
$> printf 'peace\nrest\nquiet' | sort -r
rest
quiet
peace
```

info In case you haven't noticed yet, sort adds a newline character to the final line even if it wasn't present in the input.

Numeric sort

The sort command provides various options to work with numeric formats. For most cases, the -n option is enough. Here's an example:

```
# lexicographic ordering isn't suited for numbers
$> printf '20\n2\n3\n111\n314' | sort
111
2
20
3
314
```

-n helps in this case

```
$> printf '20\n2\n3\n111\n314' | sort -n
2
3
20
111
314
```

The -n option can handle negative and floating-point numbers as well. The decimal point and the thousands separator characters will depend on the locale settings.

```
$> cat mixed_numbers.txt
12,345
42
31.24
-100
42
5678
```

, is the thousands separator in en_IN

. is the decimal point in en_IN

```
$> sort -n mixed_numbers.txt
-100
31.24
42
42
5678
12,345
```

Use the -g option if your input can have the + prefix for positive numbers or follows the E-scientific notation.

```
$> cat e_notation.txt
+120
-1.53
3.14e+4
42.1e-2
```

```
$> sort -g e_notation.txt
-1.53
42.1e-2
+120
3.14e+4
```

info Unless otherwise specified, sort will break ties by using the entire input line content. In the case of -n, sorting will work even if there are extra characters after the number. Those extra characters will affect the output order if the numbers are equal. If a line doesn't start with a number (excluding blanks), it will be treated as 0.

'b' comes before 'p'

```
$> printf '2 pins\n13 pens\n2 balls' | sort -n
2 balls
2 pins
13 pens
```

'z' and 'a2p' will be treated as '0'

'a' comes before 'z'

```
$> printf 'z\na2p\n13p\n2b\n-1\n    10' | sort -n
-1
a2p
z
2b
10
13p
```

Human numeric sort

Commands like `du` (disk usage) have the `-h` and `--si` options to display numbers with SI suffixes like `k`, `K`, `M`, `G` and so on. In such cases, you can use `sort -h` to order them.

```
$> cat file_size.txt
104K    power.log
316M    projects
746K    report.log
20K     sample.txt
1.4G    games

$> sort -hr file_size.txt
1.4G    games
316M    projects
746K    report.log
104K    power.log
20K     sample.txt
```

Version sort

The `-V` option is useful when you have a mix of alphabets and digits. It also helps when you want to treat digits after a decimal point as whole numbers, for example `1.10` should be greater than `1.2`.

```
$> printf '1.10\n1.2' | sort -n
1.10
1.2
$> printf '1.10\n1.2' | sort -V
1.2
1.10
```

```
$> cat versions.txt
file2
cmd5.2
file10
cmd1.6
file5
cmd5.10
$> sort -V versions.txt
cmd1.6
cmd5.2
cmd5.10
file2
file5
file10
```

Here's an example of dealing with numbers reported by the `time` command (assuming all the entries have the same format).

```
$> cat timings.txt
5m35.363s
3m20.058s
4m11.130s
3m42.833s
4m3.083s

$> sort -V timings.txt
3m20.058s
3m42.833s
4m3.083s
4m11.130s
5m35.363s
```

info See Version sort ordering for more details. Note that the `ls` command uses lowercase `-v` for this task.

Random sort

The `-R` option will display the output in random order. Unlike `shuf`, this option will always place identical lines next to each other due to the implementation.

```
# the two lines with '42' will always be next to each other
# use 'shuf' if you don't want this behavior
$> sort -R mixed_numbers.txt
31.24
5678
42
42
12,345
-100
```

Unique sort

The `-u` option will keep only the first copy of lines that are deemed equal.

```
# (10) and [10] are deemed equal with dictionary sorting
$> printf '(10)\n[20]\n[10]' | sort -du
(10)
[20]
```

```
$> cat purchases.txt
```



```

coffee
tea
washing powder
coffee
toothpaste
tea
soap
tea

```

```

$> sort -u purchases.txt
coffee
soap
tea
toothpaste
washing powder

```

As seen earlier, the `-n` option will work even if there are extra characters after the number. When the `-u` option is also used, only the first such copy will be retained. Use the `uniq` command if you want to remove duplicates based on the whole line.

```

$> printf '2 balls\n13 pens\n2 pins\n13 pens\n' | sort -nu
2 balls
13 pens

```

```

# note that only the output order is reversed
# use tac if you want the last duplicate to be preserved instead of the first
$> printf '2 balls\n13 pens\n2 pins\n13 pens\n' | sort -r -nu
13 pens
2 balls

```

```

# use uniq when the entire line contents should be compared
$> printf '2 balls\n13 pens\n2 pins\n13 pens\n' | sort -n | uniq
2 balls
2 pins
13 pens

```

You can use the `-f` option to ignore case while determining duplicates.

```

$> printf 'mat\nbat\nMAT\ncar\nbat\n' | sort -u
bat
car
mat
MAT

```

```

# the first copy between 'mat' and 'MAT' is retained
$> printf 'mat\nbat\nMAT\ncar\nbat\n' | sort -fu
bat
car
mat

```

Column sort

The `-k` option allows you to sort based on specific columns instead of the entire input line. By default, the empty string between non-blank and blank characters is considered as the separator and thus the blanks are also part of the field contents. The effect of blanks and mitigation will be discussed later.

The `-k` option accepts arguments in various ways. You can specify the starting and ending column numbers separated by a comma. If you specify only the starting column, the last column will be used as the ending column. Usually you just want to sort by a single column, in which case the same number is specified as both the starting and ending columns. Here's an example:

```

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

```

```

# sort based on the 2nd column numbers
$> sort -k2,2n shopping.txt
Pizza  2
toys   5
Banana 10
mango  25
apple  50

```

info Note that in the above example, the `-n` option was also appended to the `-k` option. This makes it specific to that column and overrides global options, if any. Also, remember that the entire line will be used to break ties, unless otherwise specified.

You can use the `-t` option to specify a single byte character as the field separator. Use `\0` to specify NUL as the separator. Depending on your shell you can use ANSI-C quoting to use escapes like `\t` instead of a literal tab character. When the `-t` option is used, the field separator won't be part of the field contents.

```

# department,name,marks
$> cat marks.csv
ECE,Raj,53

```

```
ECE,Joel,72
EEE,Moi,68
CSE,Surya,81
EEE,Raj,88
CSE,Moi,62
EEE,Tia,72
ECE,Om,92
CSE,Amy,67
```

```
# name column is the primary sort key
# entire line content will be used for breaking ties
$> sort -t, -k2,2 marks.csv
```

```
CSE,Amy,67
ECE,Joel,72
CSE,Moi,62
EEE,Moi,68
ECE,Om,92
ECE,Raj,53
EEE,Raj,88
CSE,Surya,81
EEE,Tia,72
```

You can use the `-k` option multiple times to specify your own order of tie breakers. Entire line will still be used to break ties if needed.

```
# second column is the primary key
# reversed numeric sort on the third column is the secondary key
# entire line will be used only if there are still tied entries
$> sort -t, -k2,2 -k3,3nr marks.csv
```

```
CSE,Amy,67
ECE,Joel,72
EEE,Moi,68
CSE,Moi,62
ECE,Om,92
EEE,Raj,88
ECE,Raj,53
CSE,Surya,81
EEE,Tia,72
```

```
# sort by month first and then the day
# -M option sorts based on abbreviated month names
$> printf 'Aug-20\nMay-5\nAug-3' | sort -t- -k1,1M -k2,2n
May-5
Aug-3
Aug-20
```

Use the `-s` option to retain the original order of input lines when two or more lines are deemed equal. You can still use multiple keys to specify your own tie breakers, `-s` only prevents the last resort comparison.

```
# -s prevents last resort comparison
# so, lines having the same value in the 2nd column will retain input order
$> sort -t, -s -k2,2 marks.csv
```

```
CSE,Amy,67
ECE,Joel,72
EEE,Moi,68
CSE,Moi,62
ECE,Om,92
ECE,Raj,53
EEE,Raj,88
CSE,Surya,81
EEE,Tia,72
```

The `-u` option, as discussed earlier, will retain only the first copy of lines that are deemed equal.

```
# only the first copy of duplicates in the 2nd column will be retained
$> sort -t, -u -k2,2 marks.csv
```

```
CSE,Amy,67
ECE,Joel,72
EEE,Moi,68
ECE,Om,92
ECE,Raj,53
CSE,Surya,81
EEE,Tia,72
```

Character positions within columns

The `-k` option also accepts starting and ending character positions within the columns. These are specified after the column number, separated by a `.` character. If the character position is not specified for the ending column, the last character of that column is assumed.

The character positions start with 1 for the first character. Recall that when the `-t` option is used, the field separator is not part of the field contents.

```
# based on the second column number
# 2.2 helps to ignore first character, otherwise -n won't have any effect here
$> printf 'car,(20)\njeep,[10]\ntruck,(5)\nbus,[3]' | sort -t, -k2.2,2n
bus,[3]
```

```
truck,(5)
jeep,[10]
car,(20)
```

```
# first character of the second column is the primary key
# entire line acts as the last resort tie breaker
$> printf 'car,(20)\njeep,[10]\ntruck,(5)\nbus,[3]' | sort -t, -k2.1,2.1
car,(20)
truck,(5)
bus,[3]
jeep,[10]
```

The default separation based on blank characters works differently. The empty string between non-blank and blank characters is considered as the separator and thus the blanks are also part of the field contents. You can use the `-b` option to ignore such leading blanks of field contents.

```
# the second column here starts with blank characters
# adjusting the character position isn't feasible due to varying blanks
$> printf 'car (20)\njeep [10]\ntruck (5)\nbus [3]' | sort -k2.2,2n
bus [3]
car (20)
jeep [10]
truck (5)
```

```
# use -b in such cases to ignore the leading blanks
$> printf 'car (20)\njeep [10]\ntruck (5)\nbus [3]' | sort -k2.2b,2n
bus [3]
truck (5)
jeep [10]
car (20)
```

Debugging

The `--debug` option can help you identify issues if the output isn't what you expected. Here's the previously seen `-b` example, now with `--debug` enabled. The underscores in the debug output shows which portions of the input are used as primary key, secondary key and so on. The collating order being used is also shown in the output.

```
$> printf 'car (20)\njeep [10]\ntruck (5)\nbus [3]' | sort -k2.2,2n --debug
sort: text ordering performed using 'en_IN' sorting rules
sort: leading blanks are significant in key 1; consider also specifying 'b'
sort: note numbers use '.' as a decimal point in this locale
bus [3]
  ^ no match for key
```

```
car (20)
  ^ no match for key
```

```
jeep [10]
  ^ no match for key
```

```
truck (5)
  ^ no match for key
```

```
$> printf 'car (20)\njeep [10]\ntruck (5)\nbus [3]' | sort -k2.2b,2n --debug
sort: text ordering performed using 'en_IN' sorting rules
sort: note numbers use '.' as a decimal point in this locale
bus [3]
```

```

truck (5)
```

```
jeep [10]
```

```
car (20)
```

Check if sorted

The `-c` option helps you spot the first unsorted entry in the given input. The uppercase `-C` option is similar but only affects the exit status. Note that these options will not work for multiple inputs.

```
$> cat shopping.txt
```

```
apple  50
toys    5
Pizza   2
mango  25
Banana  10
```

```
$> sort -c shopping.txt
sort: shopping.txt:3: disorder: Pizza  2
$> echo $?
1
```

```
$> sort -C shopping.txt
$> echo $?
1
```

Specifying output file

The `-o` option can be used to specify the output file to be used for saving the results.

```
$> sort -R nums.txt -o rand_nums.txt
```

```
$> cat rand_nums.txt
1000
3.14
42
```

You can use `-o` for in-place editing as well, but the documentation gives this warning:

However, it is often safer to output to an otherwise-unused file, as data may be lost if the system crashes or sort encounters an I/O or other serious error while a file is being sorted in place. Also, sort with `--merge (-m)` can open the output file before reading all input, so a command like `cat F | sort -m -o F - G` is not safe as sort might start writing F before cat is done reading it.

Merge sort

The `-m` option is useful if you have one or more sorted input files and need a single sorted output file. Typically the use case is that you want to add newly obtained data to existing sorted data. In such cases, you can sort only the new data separately and then combine all the sorted inputs using the `-m` option. Here's a sample timing comparison between different combinations of sorted/unordered inputs.

```
$> shuf -n1000000 -i1-999999999999 > n1.txt
$> shuf -n1000000 -i1-999999999999 > n2.txt
$> sort -n n1.txt > n1_sorted.txt
$> sort -n n2.txt > n2_sorted.txt

$> time sort -n n1.txt n2.txt > op1.txt
real    0m1.010s
$> time sort -mn n1_sorted.txt <(sort -n n2.txt) > op2.txt
real    0m0.535s
$> time sort -mn n1_sorted.txt n2_sorted.txt > op3.txt
real    0m0.218s

$> diff -sq op1.txt op2.txt
Files op1.txt and op2.txt are identical
$> diff -sq op1.txt op3.txt
Files op1.txt and op3.txt are identical

$> rm n{1,2}{,_sorted}.txt op{1..3}.txt
```

info You might wonder if you can improve the performance of a single large file using the `-m` option. By default, sort already uses the available processors to split the input and merge. You can use the `--parallel` option to customize this behavior.

NUL separator

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

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

Further Reading

A few options like `--compress-program` and `--files0-from` aren't covered in this book. See the sort manual for details and examples. See also:

- * [unix.stackexchange](#): Scalability of sort for gigantic files
- * [stackoverflow](#): Sort by last field when the number of fields varies
- * [Arch wiki](#): locale
- * [ShellHacks](#): locale and language settings

Exercises

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

1) Default sort doesn't work for numbers. Which option would you use to get the expected output shown below?

```
$> printf '100\n10\n20\n3000\n2.45\n' | sort ##### add your solution here
2.45
10
20
100
3000
```

2) Which sort option will help you ignore case? `LC_ALL=C` is used here to avoid differences due to locale.

```
$> printf 'Super\nover\nRUNE\ntea\n' | LC_ALL=C sort ##### add your solution here
over
RUNE
Super
```

tea

3) The -n option doesn't work for all sorts of numbers. Which sort option would you use to get the expected output shown below?

wrong output

```
$> printf '+120\n-1.53\n3.14e+4\n42.1e-2' | sort -n
-1.53
+120
3.14e+4
42.1e-2
```

expected output

```
$> printf '+120\n-1.53\n3.14e+4\n42.1e-2' | sort ##### add your solution here
-1.53
42.1e-2
+120
3.14e+4
```

4) What do the -V and -h options do?

5) Is there a difference between shuf and sort -R?

6) Sort the scores.csv file numerically in ascending order using the contents of the second field. Header line should be preserved as the first line as shown below.

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

add your solution here

```
Name,Maths,Physics,Chemistry
Lin,78,83,80
Cy,97,98,95
Ith,100,100,100
```

7) Sort the contents of duplicates.csv by the fourth column numbers in descending order. Retain only the first copy of lines with the same number.

```
$> cat duplicates.csv
brown,toy,bread,42
dark red,ruby,rose,111
blue,ruby,water,333
dark red,sky,rose,555
yellow,toy,flower,333
white,sky,bread,111
light red,purse,rose,333
```

add your solution here

```
dark red,sky,rose,555
blue,ruby,water,333
dark red,ruby,rose,111
brown,toy,bread,42
```

8) Sort the contents of duplicates.csv by the third column item. Use the fourth column numbers as the tie-breaker.

add your solution here

```
brown,toy,bread,42
white,sky,bread,111
yellow,toy,flower,333
dark red,ruby,rose,111
light red,purse,rose,333
dark red,sky,rose,555
blue,ruby,water,333
```

9) What does the -s option provide?

10) Sort the given input based on the numbers inside the brackets.

```
$> printf '(-3.14)\n[45]\n(12.5)\n{14093}' | ##### add your solution here
(-3.14)
(12.5)
[45]
{14093}
```

11) What do the -c, -C and -m options do?

```
---
uniq
```

The uniq command identifies similar lines that are adjacent to each other. There are various options to help you filter unique or duplicate lines, count them, group them, etc.

Retain single copy of duplicates

This is the default behavior of the `uniq` command. If adjacent lines are the same, only the first copy will be displayed in the output.

```
# only the adjacent lines are compared to determine duplicates
# which is why you get 'red' twice in the output for this input
$> printf 'red\nred\nred\ngreen\nred\nblue\nblue' | uniq
red
green
red
blue
```

You'll need sorted input to make sure all the input lines are considered to determine duplicates. For some cases, `sort -u` is enough, like the example shown below:

```
# same as sort -u for this case
$> printf 'red\nred\nred\ngreen\nred\nblue\nblue' | sort | uniq
blue
green
red
```

Sometimes though, you may need to sort based on some specific criteria and then identify duplicates based on the entire line contents. Here's an example:

```
# can't use sort -n -u here
$> printf '2 balls\n13 pens\n2 pins\n13 pens' | sort -n | uniq
2 balls
2 pins
13 pens
```

`info sort+uniq` won't be suitable if you need to preserve the input order as well. You can use alternatives like `awk`, `perl` and `huniq` for such cases.

```
# retain only the first copy of duplicates, maintain input order
$> printf 'red\nred\nred\ngreen\nred\nblue\nblue' | awk '!seen[$0]++'
red
green
blue
```

Duplicates only

The `-d` option will display only the duplicate entries. That is, only if a line is seen more than once.

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

$> sort purchases.txt | uniq -d
coffee
tea
```

To display all the copies of duplicates, use the `-D` option.

```
$> sort purchases.txt | uniq -D
coffee
coffee
tea
tea
tea
```

Unique only

The `-u` option will display only the unique entries. That is, only if a line doesn't occur more than once.

```
$> sort purchases.txt | uniq -u
soap
toothpaste
washing powder
```

```
# reminder that uniq works based on adjacent lines only
$> printf 'red\nred\nred\ngreen\nred\nblue\nblue' | uniq -u
green
red
```

Grouping similar lines

The `--group` options allows you to visually separate groups of similar lines with an empty line. This option can accept four values - `separate`, `prepend`, `append` and `both`. The default is `separate`, which adds a newline character between the groups. `prepend` will add a newline before the first group as well and `append` will add a newline after the last group. `both` combines the `prepend` and `append` behavior.

```
$> sort purchases.txt | uniq --group
coffee
coffee
```

```
soap
```

```
tea
tea
tea
```

```
toothpaste
```

```
washing powder
```

The `--group` option cannot be used with the `-c`, `-d`, `-D` or `-u` options. The `--all-repeated` alias for the `-D` option uses none as the default grouping. You can change that to separate or prepend values.

```
$> sort purchases.txt | uniq --all-repeated=prepend
```

```
coffee
coffee
```

```
tea
tea
tea
```

Prefix count

If you want to know how many times a line has been repeated, use the `-c` option. This will be added as a prefix.

```
$> sort purchases.txt | uniq -c
      2 coffee
      1 soap
      3 tea
      1 toothpaste
      1 washing powder
```

```
$> sort purchases.txt | uniq -dc
      2 coffee
      3 tea
```

The output of this option is usually piped to `sort` for ordering the output based on the count.

```
$> sort purchases.txt | uniq -c | sort -n
      1 soap
      1 toothpaste
      1 washing powder
      2 coffee
      3 tea
```

```
$> sort purchases.txt | uniq -c | sort -nr
      3 tea
      2 coffee
      1 washing powder
      1 toothpaste
      1 soap
```

Ignoring case

Use the `-i` option to ignore case while determining duplicates.

depending on your locale, `sort` and `sort -f` can give the same results

```
$> printf 'hat\nbat\nHAT\ncar\nbat\nmat\nmoat' | sort -f | uniq -iD
bat
bat
hat
HAT
```

Partial match

`uniq` has three options to change the matching criteria to partial parts of the input line. These aren't as powerful as the `sort -k` option, but they do come in handy for some use cases.

The `-f` option allows you to skip the first N fields. Field separation is based on one or more space/tab characters only. Note that these separators will still be part of the field contents, so this will not work with variable number of blanks.

skip the first field, works as expected since the no. of blanks is consistent

```
$> printf '2 cars\n5 cars\n10 jeeps\n5 jeeps\n3 trucks\n' | uniq -f1 --group
2 cars
5 cars
```

```
10 jeeps
5 jeeps
```

```
3 trucks
```

example with variable number of blanks

```
# 'cars' entries were identified as duplicates, but not 'jeeps'
$> printf '2 cars\n5 cars\n1 jeeps\n5 jeeps\n3 trucks\n' | uniq -f1
2 cars
1 jeeps
5 jeeps
3 trucks
```

The -s option allows you to skip the first N characters (calculated as bytes).

```
# skip the first character
$> printf '* red\n- green\n* green\n* blue\n= blue' | uniq -s1
* red
- green
* blue
```

The -w option restricts the comparison to the first N characters (calculated as bytes).

```
# compare only the first 2 characters
$> printf '1) apple\n1) almond\n2) banana\n3) cherry' | uniq -w2
1) apple
2) banana
3) cherry
```

When these options are used simultaneously, the priority is -f first, then -s and finally the -w option. Remember that blanks are part of the field content.

```
# skip the first field
# then skip the first two characters (including the blank character)
# use the next two characters for comparison ('bl' and 'ch' in this example)
$> printf '2 @blue\n10 :black\n5 :cherry\n3 @chalk' | uniq -f1 -s2 -w2
2 @blue
5 :cherry
```

info If a line doesn't have enough fields or characters to satisfy the -f and -s options respectively, a null string is used for comparison.

Specifying output file

uniq can accept filename as the source of input contents, but only a maximum of one file. If you specify another file, it will be used as the output file.

```
$> printf 'apple\napple\nbanana\ncherry\ncherry\ncherry' > ip.txt
$> uniq ip.txt op.txt
```

```
$> cat op.txt
apple
banana
cherry
```

NUL separator

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

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

info If grouping is specified, NUL will be used as the separator instead of the newline character.

Alternatives

Here are some alternate commands you can explore if uniq isn't enough to solve your task.

- * Dealing with duplicates chapter from my GNU awk ebook
- * Dealing with duplicates chapter from my Perl one-liners ebook
- * huniq - remove duplicates from entire input contents, input order is maintained, supports count option as well

Exercises

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

1) Will uniq throw an error if the input is not sorted? What do you think will be the output for the following input?

```
$> printf 'red\nred\nred\ngreen\nred\nblue\nblue' | uniq
```

2) Are there differences between sort -u file and sort file | uniq?

3) What are the differences between sort -u and uniq -u options, if any?

4) Filter the third column items from duplicates.csv. Construct three solutions to display only unique items, duplicate items and all duplicates.

```
$> cat duplicates.csv
brown,toy,bread,42
dark red,ruby,rose,111
blue,ruby,water,333
dark red,sky,rose,555
yellow,toy,flower,333
```



```
white,sky,bread,111
light red,purse,rose,333
```

```
# unique
##### add your solution here
flower
water
```

```
# duplicates
##### add your solution here
bread
rose
```

```
# all duplicates
##### add your solution here
bread
bread
rose
rose
rose
```

5) What does the `--group` option do? What customization features are available?

6) Count the number of times input lines are repeated and display the results in the format shown below.

```
$> s='brown\nbrown\nbrown\ngreen\nbrown\nblue\nblue'
$> printf '%b' "$s" | ##### add your solution here
    1 green
    2 blue
    4 brown
```

7) For the input file `f1.txt`, retain only unique entries based on the first two characters of each line. For example, `abcd` and `ab12` should be considered as duplicates and neither of them will be part of the output.

```
$> cat f1.txt
3) cherry
1) apple
2) banana
1) almond
4) mango
2) berry
3) chocolate
1) apple
5) cherry
```

```
##### add your solution here
4) mango
5) cherry
```

8) For the input file `f1.txt`, display only the duplicate items without considering the first two characters of each line. For example, `abcd` and `12cd` should be considered as duplicates. Assume that the third character of each line is always a space character.

```
##### add your solution here
1) apple
3) cherry
```

9) What does the `-s` option do?

10) Filter only unique lines, but ignore differences due to case.

```
$> printf 'cat\nbat\nCAT\nCar\nBat\nmat\nMat' | ##### add your solution here
Car
```

```
---
comm
```

The `comm` command finds common and unique lines between two sorted files. These results are formatted as a table with three columns and one or more of these columns can be suppressed as required.

Three column output
Consider the sample input files as shown below:

```
# side by side view of the sample files
# note that these files are already sorted
$> paste colors_1.txt colors_2.txt
Blue    Black
Brown   Blue
Orange  Green
Purple  Orange
Red     Pink
Teal    Red
White   White
```

By default, `comm` gives a tabular output with three columns:

- * first column has lines unique to the first file
- * second column has lines unique to the second file
- * third column has lines common to both the files

The columns are separated by a tab character. Here's the output for the above sample files:

```
$> comm colors_1.txt colors_2.txt
Black
      Blue
Brown
      Green
      Orange
      Pink
Purple
      Red
Teal
      White
```

You can change the column separator to a string of your choice using the `--output-delimiter` option. Here's an example:

```
# note that the input files need not have the same number of lines
$> comm <(seq 3) <(seq 2 5)
1
      2
      3
      4
      5
```

```
$> comm --output-delimiter=, <(seq 3) <(seq 2 5)
1
,,2
,,3
,4
,5
```

info Collating order for `comm` should be same as the one used to sort the input files.

info `--nocheck-order` option can be used for unsorted inputs. However, as per the documentation, this option "is not guaranteed to produce any particular output."

Suppressing columns

You can use one or more of the following options to suppress columns:

- * `-1` to suppress the lines unique to the first file
- * `-2` to suppress the lines unique to the second file
- * `-3` to suppress the lines common to both the files

Here's how the output looks like when you suppress one of the columns:

```
# suppress lines common to both the files
$> comm -3 colors_1.txt colors_2.txt
Black
Brown
      Green
      Pink
Purple
Teal
```

Combining two of these options gives three useful solutions. `-12` will give you only the common lines.

```
$> comm -12 colors_1.txt colors_2.txt
Blue
Orange
Red
White
```

`-23` will give you the lines unique to the first file.

```
$> comm -23 colors_1.txt colors_2.txt
Brown
Purple
Teal
```

`-13` will give you the lines unique to the second file.

```
$> comm -13 colors_1.txt colors_2.txt
Black
Green
Pink
```

You can combine all the three options as well. Useful with the `--total` option to get only the count of lines for each of the three columns.

```
$> comm --total -123 colors_1.txt colors_2.txt
```

```
3      3      4      total
```

Duplicate lines

The number of duplicate lines in the common column will be minimum of the duplicate occurrences between the two files. Rest of the duplicate lines, if any, will be considered as unique to the file having the excess lines. Here's an example:

```
$> paste list_1.txt list_2.txt
```

```
apple  cherry
banana cherry
cherry mango
cherry papaya
cherry
cherry
```

```
# 'cherry' occurs only twice in the second file
# rest of the 'cherry' lines will be unique to the first file
```

```
$> comm list_1.txt list_2.txt
```

```
apple
banana
                cherry
                cherry
cherry
cherry
                mango
                papaya
```

NUL separator

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

```
$> comm -z -12 <(printf 'a\0b\0c') <(printf 'a\0c\0x') | cat -v
```

```
a^@c^@
```

Alternatives

Here are some alternate commands you can explore if `comm` isn't enough to solve your task. These alternatives do not require the input files to be sorted.

- * `zet` - set operations on one or more input files
- * Comparing lines between files section from my GNU `grep` ebook
- * Two file processing chapter from my GNU `awk` ebook, has examples for both line and field based comparisons
- * Two file processing chapter from my Perl one-liners ebook, has examples for both line and field based comparisons

Exercises

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

1) Get the common lines between the `s1.txt` and `s2.txt` files. Assume that their contents are already sorted.

```
$> paste s1.txt s2.txt
```

```
apple  banana
coffee coffee
fig     eclair
honey   fig
mango   honey
pasta   milk
sugar    tea
tea      yeast
```

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

```
coffee
fig
honey
tea
```

2) Display lines present in `s1.txt` but not `s2.txt` and vice versa.

```
# lines unique to the first file
```

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

```
apple
mango
pasta
sugar
```

```
# lines unique to the second file
```

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

```
banana
eclair
milk
yeast
```

3) Display lines unique to the `s1.txt` file and the common lines when compared to the `s2.txt` file. Use `=>` to separate the output columns.

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

```
apple
==>coffee
==>fig
==>honey
mango
pasta
sugar
==>tea
```

4) What does the --total option do?

5) Will the comm command fail if there are repeated lines in the input files? If not, what'd be the expected output for the command shown below?

```
$> cat s3.txt
apple
apple
guava
honey
tea
tea
tea

$> comm -23 s3.txt s1.txt
```

```
---
join
```

The join command helps you to combine lines from two files based on a common field. This works best when the input is already sorted by that field.

Default join

By default, join combines two files based on the first field content (also referred as key). Only the lines with common keys will be part of the output.

The key field will be displayed first in the output (this distinction will come into play if the first field isn't the key). Rest of the line will have the remaining fields from the first and second files, in that order. One or more blanks (space or tab) will be considered as the input field separator and a single space will be used as the output field separator. If present, blank characters at the start of the input lines will be ignored.

```
# sample sorted input files
$> cat shopping_jan.txt
apple 10
banana 20
soap 3
tshirt 3
$> cat shopping_feb.txt
banana 15
fig 100
pen 2
soap 1
```

```
# combine common lines based on the first field
$> join shopping_jan.txt shopping_feb.txt
banana 20 15
soap 3 1
```

If a field value is present multiple times in the same input file, all possible combinations will be present in the output. As shown below, join will also ensure to add a final newline character even if it wasn't present in the input.

```
$> join <(printf 'a f1_x\n a f1_y') <(printf 'a f2_x\n a f2_y')
a f1_x f2_x
a f1_x f2_y
a f1_y f2_x
a f1_y f2_y
```

info Note that the collating order used for join should be same as the one used to sort the input files. Use join -i to ignore case, similar to sort -f usage.

info If the input files are not sorted, join will produce an error if there are unpairable lines. You can use the --nocheck-order option to ignore this error. However, as per the documentation, this option "is not guaranteed to produce any particular output."

Non-matching lines

By default, only the lines having common keys are part of the output. You can use the -a option to also include the non-matching lines from the input files. Use 1 and 2 as the argument for the first and second file respectively. You'll later see how to fill missing fields with a custom string.

```
# includes non-matching lines from the first file
$> join -a1 shopping_jan.txt shopping_feb.txt
apple 10
banana 20 15
soap 3 1
```

```
tshirt 3
```

```
# includes non-matching lines from both the files
$> join -a1 -a2 shopping_jan.txt shopping_feb.txt
apple 10
banana 20 15
fig 100
pen 2
soap 3 1
tshirt 3
```

If you use `-v` instead of `-a`, the output will have only the non-matching lines.

```
$> join -v2 shopping_jan.txt shopping_feb.txt
fig 100
pen 2
```

```
$> join -v1 -v2 shopping_jan.txt shopping_feb.txt
apple 10
fig 100
pen 2
tshirt 3
```

Change field separator

You can use the `-t` option to specify a single byte character as the field separator. The output field separator will be same as the value used for the `-t` option. Use `\0` to specify NUL as the separator. Empty string will cause entire input line content to be considered as keys. Depending on your shell you can use ANSI-C quoting to use escapes like `\t` instead of a literal tab character.

```
$> cat marks.csv
ECE,Raj,53
ECE,Joel,72
EEE,Moi,68
CSE,Surya,81
EEE,Raj,88
CSE,Moi,62
EEE,Tia,72
ECE,Om,92
CSE,Amy,67
$> cat dept.txt
CSE
ECE
```

```
# get all lines from marks.csv based on the first field keys in dept.txt
$> join -t, <(sort marks.csv) dept.txt
CSE,Amy,67
CSE,Moi,62
CSE,Surya,81
ECE,Joel,72
ECE,Om,92
ECE,Raj,53
```

Files with headers

Use the `--header` option to ignore first lines of both the input files from sorting consideration. Without this option, the `join` command might still work correctly if unpairable lines aren't found, but it is preferable to use `--header` when applicable. This option will also help when `--check-order` option is active.

```
$> cat report_1.csv
Name,Maths,Physics
Amy,78,95
Moi,88,75
Raj,67,76
$> cat report_2.csv
Name,Chemistry
Amy,85
Joel,78
Raj,72

$> join --check-order -t, report_1.csv report_2.csv
join: report_1.csv:2: is not sorted: Amy,78,95
$> join --check-order --header -t, report_1.csv report_2.csv
Name,Maths,Physics,Chemistry
Amy,78,95,85
Raj,67,76,72
```

Change key field

By default, the first field of both the input files are used to combine the lines. You can use `-1` and `-2` options followed by a field number to specify a different field number. You can use the `-j` option if the field number is the same for both the files.

Recall that the key field is the first field in the output. You'll later see how to customize the output field order.

```
$> cat names.txt
Amy
Raj
```

Tia

```
# combine based on the second field of the first file
# and the first field of the second file (default)
$> join -t, -1 2 <(sort -t, -k2,2 marks.csv) names.txt
Amy,CSE,67
Raj,ECE,53
Raj,EEE,88
Tia,EEE,72
```

Customize output field list

Use the -o option to customize the fields required in the output and their order. Especially useful when the first field isn't the key. Each output field is specified as file number followed by a . character and then the field number. You can specify multiple fields separated by a , character. As a special case, you can use 0 to indicate the key field.

```
# output field order is 1st, 2nd and 3rd fields from the first file
$> join -t, -1 2 -o 1.1,1.2,1.3 <(sort -t, -k2,2 marks.csv) names.txt
CSE,Amy,67
ECE,Raj,53
EEE,Raj,88
EEE,Tia,72
```

```
# 1st field from the first file, 2nd field from the second file
# and then 2nd and 3rd fields from the first file
$> join --header -t, -o 1.1,2.2,1.2,1.3 report_1.csv report_2.csv
Name,Chemistry,Maths,Physics
Amy,85,78,95
Raj,72,67,76
```

Same number of output fields

If you use auto as the argument for the -o option, first line of both the input files will be used to determine the number of output fields. If the other lines have extra fields, they will be discarded.

```
$> join <(printf 'a 1 2\nb p q r') <(printf 'a 3 4\nb x y z')
a 1 2 3 4
b p q r x y z
```

```
$> join -o auto <(printf 'a 1 2\nb p q r') <(printf 'a 3 4\nb x y z')
a 1 2 3 4
b p q x y
```

If the other lines have lesser number of fields, the -e option will determine the string to be used as a filler (empty string is the default).

```
# the second line has two empty fields
$> join -o auto <(printf 'a 1 2\nb p') <(printf 'a 3 4\nb x')
a 1 2 3 4
b p x
```

```
$> join -o auto -e '-' <(printf 'a 1 2\nb p') <(printf 'a 3 4\nb x')
a 1 2 3 4
b p - x -
```

As promised earlier, here are some examples of filling fields for non-matching lines:

```
$> join -o auto -a1 -e 'NA' shopping_jan.txt shopping_feb.txt
apple 10 NA
banana 20 15
soap 3 1
tshirt 3 NA
```

```
$> join -o auto -a1 -a2 -e 'NA' shopping_jan.txt shopping_feb.txt
apple 10 NA
banana 20 15
fig NA 100
pen NA 2
soap 3 1
tshirt 3 NA
```

Set operations

This section covers whole line set operations you can perform on already sorted input files. Equivalent sort and uniq solutions will also be mentioned as comments (useful for unsorted inputs). Assume that there are no duplicate lines within an input file.

These two sorted input files will be used for the examples to follow:

```
$> paste colors_1.txt colors_2.txt
Blue    Black
Brown   Blue
Orange  Green
Purple  Orange
Red      Pink
Teal     Red
White    White
```

Here's how you can get union and symmetric difference results. Recall that `-t ''` will cause the entire input line content to be considered as keys.

```
# union
# unsorted input: sort -u colors_1.txt colors_2.txt
$> join -t '' -a1 -a2 colors_1.txt colors_2.txt
Black
Blue
Brown
Green
Orange
Pink
Purple
Red
Teal
White

# symmetric difference
# unsorted input: sort colors_1.txt colors_2.txt | uniq -u
$> join -t '' -v1 -v2 colors_1.txt colors_2.txt
Black
Brown
Green
Pink
Purple
Teal
```

Here's how you can get intersection and difference results. The equivalent `comm` solutions for sorted input is also mentioned in the comments.

```
# intersection, same as: comm -12 colors_1.txt colors_2.txt
# unsorted input: sort colors_1.txt colors_2.txt | uniq -d
$> join -t '' colors_1.txt colors_2.txt
Blue
Orange
Red
White

# difference, same as: comm -13 colors_1.txt colors_2.txt
# unsorted input: sort colors_1.txt colors_2.txt | uniq -u
$> join -t '' -v2 colors_1.txt colors_2.txt
Black
Green
Pink

# difference, same as: comm -23 colors_1.txt colors_2.txt
# unsorted input: sort colors_1.txt colors_2.txt colors_2.txt | uniq -u
$> join -t '' -v1 colors_1.txt colors_2.txt
Brown
Purple
Teal
```

As mentioned before, `join` will display all the combinations if there are duplicate entries. Here's an example to show the differences between `sort`, `comm` and `join` solutions for displaying common lines:

```
$> paste list_1.txt list_2.txt
apple  cherry
banana cherry
cherry mango
cherry papaya
cherry
cherry

# only one entry per common line
$> sort list_1.txt list_2.txt | uniq -d
cherry

# minimum of 'no. of entries in file1' and 'no. of entries in file2'
$> comm -12 list_1.txt list_2.txt
cherry
cherry

# 'no. of entries in file1' multiplied by 'no. of entries in file2'
$> join -t '' list_1.txt list_2.txt
cherry
cherry
cherry
cherry
cherry
cherry
cherry
cherry
```

NUL separator

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

```
$> join -z <(printf 'a 1\0b x') <(printf 'a 2\0b y') | cat -v
a 1 2^@b x y^@
```

Alternatives

Here are some alternate commands you can explore if join isn't enough to solve your task. These alternatives do not require input to be sorted.

- * `zet` - set operations on one or more input files
- * Comparing lines between files section from my GNU `grep` ebook
- * Two file processing chapter from my GNU `awk` ebook, has examples for both line and field based comparisons
- * Two file processing chapter from my Perl one-liners ebook, has examples for both line and field based comparisons

Exercises

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

info Assume that the input files are already sorted for these exercises.

1) Use appropriate options to get the expected outputs shown below.

```
# no output
$> join <(printf 'apple 2\nfig 5') <(printf 'Fig 10\nmango 4')
```

```
# expected output 1
##### add your solution here
fig 5 10
```

```
# expected output 2
##### add your solution here
apple 2
fig 5 10
mango 4
```

2) Use the `join` command to display only the non-matching lines based on the first field.

```
$> cat j1.txt
apple 2
fig 5
lemon 10
tomato 22
$> cat j2.txt
almond 33
fig 115
mango 20
pista 42
```

```
# first field items present in j1.txt but not j2.txt
##### add your solution here
apple 2
lemon 10
tomato 22
```

```
# first field items present in j2.txt but not j1.txt
##### add your solution here
almond 33
mango 20
pista 42
```

3) Filter lines from `j1.txt` and `j2.txt` that match the items from `s1.txt`.

```
$> cat s1.txt
apple
coffee
fig
honey
mango
pasta
sugar
tea
```

```
##### add your solution here
apple 2
fig 115
fig 5
mango 20
```

4) Join the `marks_1.csv` and `marks_2.csv` files to get the expected output shown below.

```
$> cat marks_1.csv
Name,Biology,Programming
Er,92,77
Ith,100,100
Lin,92,100
Sil,86,98
$> cat marks_2.csv
```



```
Name,Maths,Physics,Chemistry
Cy,97,98,95
Ith,100,100,100
Lin,78,83,80
```

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

```
Name,Biology,Programming,Maths,Physics,Chemistry
Ith,100,100,100,100,100
Lin,92,100,78,83,80
```

5) By default, the first field is used to combine the lines. Which options are helpful if you want to change the key field to be used for joining?

6) Join the marks_1.csv and marks_2.csv files to get the expected output with specific fields as shown below.

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

```
Name,Programming,Maths,Biology
Ith,100,100,100
Lin,100,78,92
```

7) Join the marks_1.csv and marks_2.csv files to get the expected output shown below. Use 50 as the filler data.

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

```
Name,Biology,Programming,Maths,Physics,Chemistry
Cy,50,50,97,98,95
Er,92,77,50,50,50
Ith,100,100,100,100,100
Lin,92,100,78,83,80
Sil,86,98,50,50,50
```

8) When you use the -o auto option, what'd happen to the extra fields compared to those in the first lines of the input data?

9) From the input files j3.txt and j4.txt, filter only the lines are unique - i.e. lines that are not common to these files. Assume that the input files do not have duplicate entries.

```
$> cat j3.txt
```

```
almond
apple pie
cold coffee
honey
mango shake
pasta
sugar
tea
```

```
$> cat j4.txt
```

```
apple
banana shake
coffee
fig
honey
mango shake
milk
tea
yeast
```

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

```
almond
apple
apple pie
banana shake
coffee
cold coffee
fig
milk
pasta
sugar
yeast
```

10) From the input files j3.txt and j4.txt, filter only the lines are common to these files.

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

```
honey
mango shake
tea
```

```
---
```

```
nl
```

If the numbering options provided by cat isn't enough, nl might suit you better. Apart from options to customize the number formatting and the separator, you can also filter which lines should be numbered. Additionally, you can divide your input into sections and number them separately.

Default numbering

By default, `nl` will prefix line numbers and a tab character to every non-empty input lines. The default number formatting is 6 characters wide and right justified with spaces. Similar to `cat`, the `nl` command will concatenate multiple inputs.

```
# same as: cat -n greeting.txt fruits.txt nums.txt
```

```
$> nl 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
```

```
# example for input with empty lines, same as: cat -b
```

```
$> printf 'apple\n\nbanana\n\ncherry\n' | nl
```

```
1 apple

2 banana

3 cherry
```

Number formatting

You can use the `-n` option to customize the number formatting. The available styles are:

- * `rn` right justified with space fillers (default)
- * `rz` right justified with leading zeros
- * `ln` left justified with space fillers

```
# right justified with space fillers
```

```
$> nl -n'rn' greeting.txt
```

```
1 Hi there
2 Have a nice day
```

```
# right justified with leading zeros
```

```
$> nl -n'rz' greeting.txt
```

```
000001 Hi there
000002 Have a nice day
```

```
# left justified with space fillers
```

```
$> nl -n'ln' greeting.txt
```

```
1 Hi there
2 Have a nice day
```

Customize width

You can use the `-w` option to specify the width to be used for the numbers (default is 6).

```
$> nl greeting.txt
```

```
1 Hi there
2 Have a nice day
```

```
$> nl -w2 greeting.txt
```

```
1 Hi there
2 Have a nice day
```

Customize separator

By default, a tab character is used to separate the line number and the line content. You can use the `-s` option to specify your own custom string separator.

```
$> nl -w2 -s' ' greeting.txt
```

```
1 Hi there
2 Have a nice day
```

```
$> nl -w1 -s' --> ' greeting.txt
```

```
1 --> Hi there
2 --> Have a nice day
```

Starting number and step value

The `-v` option allows you to specify a different starting integer. Negative integer is also allowed.

```
$> nl -v10 greeting.txt
```

```
10 Hi there
11 Have a nice day
```

```
$> nl -v-1 fruits.txt
```

```
-1 banana
0 papaya
1 mango
```

The `-i` option allows you to specify an integer as the step value (default is 1).

```
$> nl -w2 -s') ' -i2 greeting.txt fruits.txt nums.txt
```

```
1) Hi there
3) Have a nice day
5) banana
```

```

7) papaya
9) mango
11) 3.14
13) 42
15) 1000

```

```
$> nl -w1 -s' ' -v8 -i-1 greeting.txt fruits.txt
```

```

8. Hi there
7. Have a nice day
6. banana
5. papaya
4. mango

```

Section wise numbering

If you organize your input with lines conforming to specific patterns, you can control their numbering separately. nl recognizes three types of sections with the following default patterns:

```

* \: \: as header
* \: as body
* \: as footer

```

These special lines will be replaced with an empty line after numbering. The numbering will be reset at the start of every section. Here's an example with multiple body sections:

```
$> cat body.txt
```

```

\:\:
Hi there
How are you
\:\:
banana
papaya
mango

```

```
$> nl -w1 -s' ' body.txt
```

```

1 Hi there
2 How are you

```

```

1 banana
2 papaya
3 mango

```

Here's an example with both header and body sections. By default, header and footer section lines are not numbered (you'll see options to enable them later).

```
$> cat header_body.txt
```

```

\:\:\:
Header
teal
\:\:
Hi there
How are you
\:\:
banana
papaya
mango
\:\:\:
Header
green

```

```
$> nl -w1 -s' ' header_body.txt
```

```

Header
teal

1 Hi there
2 How are you

1 banana
2 papaya
3 mango

```

```

Header
green

```

And here's an example with all the three types of sections:

```
$> cat all_sections.txt
```

```

\:\:\:
Header
teal
\:\:
Hi there
How are you
\:\:
banana

```

```
papaya
mango
\:
```

```
Footer
```

```
$> nl -w1 -s' ' all_sections.txt
```

```
Header
teal

1 Hi there
2 How are you

1 banana
2 papaya
3 mango

Footer
```

The -b, -h and -f options control which lines should be numbered for the three types of sections. Use a to number all lines of a particular section (other features will discussed later).

```
$> nl -w1 -s' ' -ha -fa all_sections.txt
```

```
1 Header
2 teal

1 Hi there
2 How are you

1 banana
2 papaya
3 mango

1 Footer
```

If you use the -p option, the numbering will not be reset on encountering a new section.

```
$> nl -p -w1 -s' ' all_sections.txt
```

```
Header
teal

1 Hi there
2 How are you

3 banana
4 papaya
5 mango

Footer
```

```
$> nl -p -w1 -s' ' -ha -fa all_sections.txt
```

```
1 Header
2 teal

3 Hi there
4 How are you

5 banana
6 papaya
7 mango

8 Footer
```

The -d option allows you to customize the two character pattern used for sections.

```
# pattern changed from \: to %=
```

```
$> cat body_sep.txt
```

```
%=
apple
banana
%=
teal
green
```

```
$> nl -w1 -s' ' -d'%= ' body_sep.txt
```

```
1 apple
2 banana

1 teal
2 green
```

Section numbering criteria

As mentioned earlier, the `-b`, `-h` and `-f` options control which lines should be numbered for the three types of sections. These options accept the following arguments:

- * `a` number all lines, including empty lines
- * `t` number lines except empty ones (default for body sections)
- * `n` do not number lines (default for header and footer sections)
- * `pBRE` use basic regular expressions (BRE) to filter lines for numbering

If the input doesn't have special patterns to identify the different sections, it will be treated as if it has a single body section. Here's an example to include empty lines for numbering:

```
$> printf 'apple\n\nbanana\n\ncherry\n' | nl -w1 -s' ' -ba
1 apple
2
3 banana
4
5 cherry
```

The `-l` option controls how many consecutive empty lines should be considered as a single entry. Only the last empty line of such groupings will be numbered.

```
# only the 2nd consecutive empty line will be considered for numbering
$> printf 'a\n\n\n\nb\n\nc' | nl -w1 -s' ' -ba -l2
1 a
2
3
4 b
5 c
```

Here's an example which uses regular expressions to identify the lines to be numbered:

```
# number lines starting with 'c' or 't'
$> nl -w1 -s' ' -bp'^[ct]' purchases.txt
1 coffee
2 tea
   washing powder
3 coffee
4 toothpaste
5 tea
   soap
6 tea
```

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

Exercises

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

- 1) `nl` and `cat -n` are always equivalent for numbering lines. True or False?
- 2) What does the `-n` option do?
- 3) Use `nl` to produce the two expected outputs shown below.

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

# expected output 1
##### add your solution here
001   Hi there
002   Have a nice day
```

```
# expected output 2
##### add your solution here
001) Hi there
002) Have a nice day
```

- 4) Figure out the logic based on the given input and output data.

```
$> cat s1.txt
apple
coffee
fig
honey
mango
pasta
sugar
tea

##### add your solution here
15. apple
13. coffee
```

```

11. fig
9. honey
7. mango
5. pasta
3. sugar
1. tea

```

5) What are the three types of sections supported by nl?

6) Only number the lines that start with ---- in the format shown below.

```
$> cat blocks.txt
```

```

----
apple--banana
mango---fig

```

```

----
3.14
-42
1000

```

```

----
sky blue
dark green

```

```

----
hi hello

```

add your solution here

```

1) ----
   apple--banana
   mango---fig

```

```

2) ----
   3.14
   -42
   1000

```

```

3) ----
   sky blue
   dark green

```

```

4) ----
   hi hello

```

7) For the blocks.txt file, determine the logic to produce the expected output shown below.

add your solution here

```

1. apple--banana
2. mango---fig

```

```

1. 3.14
2. -42
3. 1000

```

```

1. sky blue
2. dark green

```

```

1. hi hello

```

8) What does the -l option do?

9) Figure out the logic based on the given input and output data.

```
$> cat all_sections.txt
```

```

\:\:\:
Header
teal
\:\:\:
Hi there
How are you
\:\:\:
banana
papaya
mango
\:\:\:
Footer

```

add your solution here

```

1) Header
2) teal

```

```

3) Hi there
4) How are you

```

```

5) banana
6) papaya
7) mango

```

```

Footer

```

```
---
wc
```

The `wc` command is useful to count the number of lines, words and characters for the given inputs.

Line, word and byte counts

By default, the `wc` command reports the number of lines, words and bytes (in that order). The byte count includes the newline characters, so you can use that as a measure of file size as well. Here's an example:

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

$> wc greeting.txt
 2  6 25 greeting.txt
```

Wondering why there are leading spaces in the output? They help in aligning results for multiple files (discussed later).

Individual counts

Instead of the three default values, you can use options to get only the particular counts you are interested in. These options are:

- * `-l` for line count
- * `-w` for word count
- * `-c` for byte count

```
$> wc -l greeting.txt
2 greeting.txt

$> wc -w greeting.txt
6 greeting.txt

$> wc -c greeting.txt
25 greeting.txt

$> wc -wc greeting.txt
 6 25 greeting.txt
```

With stdin data, you'll get only the count value (unless you use `-` for stdin). Useful for assigning the output to shell variables.

```
$> printf 'hello' | wc -c
5
$> printf 'hello' | wc -c -
5 -

$> lines=$(wc -l &lt;greeting.txt)
$> echo "$lines"
2
```

Multiple files

If you pass multiple files to the `wc` command, the count values will be displayed separately for each file. You'll also get a summary at the end, which sums the respective count of all the input files.

```
$> wc greeting.txt nums.txt purchases.txt
 2  6 25 greeting.txt
 3  3 13 nums.txt
 8  9 57 purchases.txt
13 18 95 total
$> wc greeting.txt nums.txt purchases.txt | tail -n1
13 18 95 total

$> wc *[ck]*.csv
 9  9 101 marks.csv
 4  4  70 scores.csv
13 13 171 total
```

If you have NUL separated filenames (for example, output from `find -print0`, `grep -lZ`, etc), you can use the `--files0-from` option. This option accepts a file containing the NUL separated data (use `-` for stdin).

```
$> printf 'greeting.txt\0nums.txt' | wc --files0-from=-
2 6 25 greeting.txt
3 3 13 nums.txt
5 9 38 total
```

Character count

Use the `-m` option instead of `-c` if the input has multibyte characters.

```
# byte count
$> printf 'İ±İ»İµİ\200İçİ\215' | wc -c
12

# character count
```

```
cli-text-processing-with-gnu-coreutils-20250605.txt
```

```
$> printf 'İ±İ»İμİ\200İçİ\215' | wc -m
6
```

info Note that the current locale will affect the behavior of the -m option.

```
$> printf 'İ±İ»İμİ\200İçİ\215' | LC_ALL=C wc -m
12
```

Longest line length

You can use the -L option to report the length of the longest line in the input (excluding the newline character of a line).

```
$> echo 'apple' | wc -L
5
# last line not ending with newline won't be a problem
$> printf 'apple\nbanana' | wc -L
6

$> wc -L sample.txt
26 sample.txt
$> wc -L &LTsample.txt
26
```

If multiple files are passed, the last line summary will show the maximum length among the given inputs.

```
$> wc -L greeting.txt nums.txt purchases.txt
15 greeting.txt
 4 nums.txt
14 purchases.txt
15 total
```

Corner cases

Line count is based on the number of newline characters. So, if the last line of the input doesn't end with the newline character, it won't be counted.

```
$> printf 'good\nmorning\n' | wc -l
2
```

```
$> printf 'good\nmorning' | wc -l
1
```

```
$> printf '\n\n\n' | wc -l
3
```

Word count is based on whitespace separation. You'll have to pre-process the input if you do not want certain non-whitespace characters to influence the results.

```
$> echo 'apple ; banana ; cherry' | wc -w
5
```

```
# remove characters other than alphabets and whitespaces
$> echo 'apple ; banana ; cherry' | tr -cd 'a-zA-Z[:space:]'
apple banana cherry
$> echo 'apple ; banana ; cherry' | tr -cd 'a-zA-Z[:space:]' | wc -w
3
```

```
# allow numbers as well
$> echo '2 : apples ;' | tr -cd '[:alnum:][:space:]' | wc -w
2
```

-L won't count non-printable characters and tabs are converted to equivalent spaces. Multibyte characters will each be counted as 1 (depending on the locale, they might become non-printable too).

```
# tab characters can occupy up to 8 columns
$> printf '\t' | wc -L
8
$> printf 'a\tb' | wc -L
9
```

```
# example for non-printable character
$> printf 'a\34b' | wc -L
2
```

```
# multibyte characters are counted as 1 each in supported locales
$> printf 'İ±İ»İμİ\200İçİ\215' | wc -L
6
# non-supported locales can cause them to be treated as non-printable
$> printf 'İ±İ»İμİ\200İçİ\215' | LC_ALL=C wc -L
0
```

-m and -L options count grapheme clusters differently.

```
$> printf 'cagİ\210e' | wc -m
5
```

```
$> printf 'cagİ\210e' | wc -L
```


4

Exercises

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

1) Save the number of lines in the greeting.txt input file to the lines shell variable.

```
$> lines=##### add your solution here
$> echo "$lines"
2
```

2) What do you think will be the output of the following command?

```
$> echo 'dragons:2 ; unicorns:10' | wc -w
```

3) Use appropriate options and arguments to get the output as shown below. Also, why is the line count showing as 2 instead of 3 for the stdin data?

```
$> printf 'apple\nbanana\ncherry' | ##### add your solution here
      2      25 greeting.txt
      2      19 -
      4      44 total
```

4) Use appropriate options and arguments to get the output shown below.

```
$> printf 'greeting.txt\tscores.csv' | ##### add your solution here
2 6 25 greeting.txt
4 4 70 scores.csv
6 10 95 total
```

5) What is the difference between wc -c and wc -m options? And which option would you use to get the longest line length?

6) Calculate the number of comma separated words from the scores.csv file.

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

```
##### add your solution here
16
```

```
---
split
```

The split command is useful to divide the input into smaller parts based on the number of lines, bytes, file size, etc. You can also execute another command on the divided parts before saving the results. An example use case is sending a large file as multiple parts as a workaround for online transfer size limits.

info Since a lot of output files will be generated in this chapter (often with the same filenames), remove these files after every illustration.

Default split

By default, the split command divides the input 1000 lines at a time. Newline character is the default line separator. You can pass a single file or stdin data as the input. Use cat if you need to concatenate multiple input sources.

By default, the output files will be named xaa, xab, xac and so on (where x is the prefix). If the filenames are exhausted, two more letters will be appended and the pattern will continue as needed. If the number of input lines is not evenly divisible, the last file will contain less than 1000 lines.

```
# divide input 1000 lines at a time
$> seq 10000 | split
```

```
# output filenames
$> ls x*
xaa xab xac xad xae xaf xag xah xai xaj
```

```
# preview of some of the output files
```

```
$> head -n1 xaa xab xae xaj
==> xaa <==
1
```

```
==> xab <==
1001
```

```
==> xae <==
4001
```

```
==> xaj <==
9001
```

```
$> rm x*
```

info warning As mentioned earlier, remove the output files after every illustration.

Change number of lines

You can use the -l option to change the number of lines to be saved in each output file.

maximum of 3 lines at a time

```
$> split -l3 purchases.txt
```

```
$> head x*
```

```
==> xaa <==
```

```
coffee
```

```
tea
```

```
washing powder
```

```
==> xab <==
```

```
coffee
```

```
toothpaste
```

```
tea
```

```
==> xac <==
```

```
soap
```

```
tea
```

Split by byte count

The -b option allows you to split the input by the number of bytes. Similar to line based splitting, you can always reconstruct the input by concatenating the output files. This option also accepts suffixes such as K for 1024 bytes, KB for 1000 bytes, M for 1024 * 1024 bytes and so on.

maximum of 15 bytes at a time

```
$> split -b15 greeting.txt
```

```
$> head x*
```

```
==> xaa <==
```

```
Hi there
```

```
Have a
```

```
==> xab <==
```

```
nice day
```

when you concatenate the output files, you'll the original input

```
$> cat x*
```

```
Hi there
```

```
Have a nice day
```

The -C option is similar to the -b option, but it will try to break on line boundaries if possible. The break will happen before the given byte limit. Here's an example where input lines do not exceed the given byte limit:

```
$> split -C20 purchases.txt
```

```
$> head x*
```

```
==> xaa <==
```

```
coffee
```

```
tea
```

```
==> xab <==
```

```
washing powder
```

```
==> xac <==
```

```
coffee
```

```
toothpaste
```

```
==> xad <==
```

```
tea
```

```
soap
```

```
tea
```

```
$> wc -c x*
```

```
11 xaa
```

```
15 xab
```

```
18 xac
```

```
13 xad
```

```
57 total
```

If a line exceeds the given limit, it will be broken down into multiple parts:

```
$> printf 'apple\nbanana\n' | split -C4
```

```
$> head x*
```

```
==> xaa <==
```

```
appl
```

```
==> xab <==
```

```
e
```

```
==> xac <==
```

```
bana
==> xad <==
na
```

```
$> cat x*
apple
banana
```

Divide based on file size

The -n option has several features. If you pass only a numeric argument N, the given input file will be divided into N chunks. The output files will be roughly the same size.

divide the file into 2 parts

```
$> split -n2 purchases.txt
```

```
$> head x*
```

```
==> xaa <==
```

```
coffee
tea
washing powder
```

```
co
```

```
==> xab <==
```

```
ffee
toothpaste
tea
soap
tea
```

the two output files are roughly the same size

```
$> wc x*
```

```
3  5 28 xaa
5  5 29 xab
8 10 57 total
```

warning Since the division is based on file size, stdin data cannot be used. Newer versions of the coreutils package supports this use case by creating a temporary file before splitting.

```
$> seq 6 | split -n2
split: -: cannot determine file size
```

By using K/N as the argument, you can view the Kth chunk of N parts on stdout. No output file will be created in this scenario.

divide the input into 2 parts

view only the 1st chunk on stdout

```
$> split -n1/2 greeting.txt
```

```
Hi there
```

```
Hav
```

To avoid splitting a line, use l/ as a prefix. Quoting from the manual:

For l mode, chunks are approximately input size / N. The input is partitioned into N equal sized portions, with the last assigned any excess. If a line starts within a partition it is written completely to the corresponding file. Since lines or records are not split even if they overlap a partition, the files written can be larger or smaller than the partition size, and even empty if a line/record is so long as to completely overlap the partition.

divide input into 2 parts, but don't split lines

```
$> split -nl/2 purchases.txt
```

```
$> head x*
```

```
==> xaa <==
```

```
coffee
tea
washing powder
coffee
```

```
==> xab <==
```

```
toothpaste
tea
soap
tea
```

Here's an example to view the Kth chunk without splitting lines:

2nd chunk of 3 parts without splitting lines

```
$> split -nl/2/3 sample.txt
```

```
7) Believe it
```

```
8)
```

```
9) banana
```

```
10) papaya
```

```
11) mango
```

Interleaved lines

The -n option will also help you create output files with interleaved lines. Since this is based on the line separator and not file size, stdin data can also be used. Use the r/ prefix to enable this feature.

two parts, lines distributed in round robin fashion

```
$> seq 5 | split -nr/2
```

```
$> head x*
==> xaa <==
1
3
5
```

```
==> xab <==
2
4
```

Here's an example to view the Kth chunk:

```
$> split -nr/1/3 sample.txt
1) Hello World
4) How are you
7) Believe it
10) papaya
13) Much ado about nothing
```

Custom line separator

You can use the `-t` option to specify a single byte character as the line separator. Use `\0` to specify NUL as the separator. Depending on your shell you can use ANSI-C quoting to use escapes like `\t` instead of a literal tab character.

```
$> printf 'apple\nbanana\n;mango\npapaya\n' | split -t';' -l1
```

```
$> head x*
==> xaa <==
apple
banana
;
==> xab <==
mango
papaya
```

Customize filenames

As seen earlier, `x` is the default prefix for output filenames. To change this prefix, pass an argument after the input source.

```
# choose prefix as 'op_' instead of 'x'
$> split -l1 greeting.txt op_
```

```
$> head op_*
==> op_aa <==
Hi there
```

```
==> op_ab <==
Have a nice day
```

The `-a` option controls the length of the suffix. You'll get an error if this length isn't enough to cover all the output files. In such a case, you'll still get output files that can fit within the given length.

```
$> seq 10 | split -l1 -a1
$> ls x*
xa  xb  xc  xd  xe  xf  xg  xh  xi  xj
$> rm x*
```

```
$> seq 10 | split -l1 -a3
$> ls x*
xaaa xaab xaac xaad xaae xaaf xaag xaah xaai xaaj
$> rm x*
```

```
$> seq 100 | split -l1 -a1
split: output file suffixes exhausted
$> ls x*
xa  xc  xe  xg  xi  xk  xm  xo  xq  xs  xu  xw  xy
xb  xd  xf  xh  xj  xl  xn  xp  xr  xt  xv  xx  xz
$> rm x*
```

You can use the `-d` option to use numeric suffixes, starting from `00` (length can be changed using the `-a` option). You can use the long option `--numeric-suffixes` to specify a different starting number.

```
$> seq 10 | split -l1 -d
$> ls x*
x00 x01 x02 x03 x04 x05 x06 x07 x08 x09
$> rm x*
```

```
$> seq 10 | split -l2 --numeric-suffixes=10
$> ls x*
x10 x11 x12 x13 x14
```

Use `-x` and `--hex-suffixes` options for hexadecimal numbering.

```
$> seq 10 | split -l1 --hex-suffixes=8
$> ls x*
x08 x09 x0a x0b x0c x0d x0e x0f x10 x11
```

You can use the `--additional-suffix` option to add a constant string at the end of filenames.

```
$> seq 10 | split -l2 -a1 --additional-suffix='.log'
$> ls x*
xa.log xb.log xc.log xd.log xe.log
$> rm x*
```

```
$> seq 10 | split -l2 -a1 -d --additional-suffix='.txt' - num_
$> ls num_*
num_0.txt num_1.txt num_2.txt num_3.txt num_4.txt
```

Exclude empty files

You can sometimes end up with empty files. For example, trying to split into more parts than possible with the given criteria. In such cases, you can use the `-e` option to prevent empty files in the output. The `split` command will ensure that the filenames are sequential even if files in the middle are empty.

'xac' is empty in this example

```
$> split -nl/3 greeting.txt
$> head x*
==> xaa <==
Hi there
```

```
==> xab <==
Have a nice day
```

```
==> xac <==
```

```
$> rm x*
```

prevent empty files

```
$> split -e -nl/3 greeting.txt
$> head x*
==> xaa <==
Hi there
```

```
==> xab <==
Have a nice day
```

Process parts through another command

The `--filter` option will allow you to apply another command on the intermediate split results before saving the output files. Use `$FILE` to refer to the output filename of the intermediate parts. Here's an example of compressing the results:

```
$> split -l1 --filter='gzip > $FILE.gz' greeting.txt
```

```
$> ls x*
xaa.gz xab.gz
```

```
$> zcat xaa.gz
Hi there
$> zcat xab.gz
Have a nice day
```

Here's an example of ignoring the first line of the results:

```
$> cat body_sep.txt
```

```
%=%=
apple
banana
%=%=
red
green
```

```
$> split -l3 --filter='tail -n +2 > $FILE' body_sep.txt
```

```
$> head x*
==> xaa <==
apple
banana
```

```
==> xab <==
red
green
```

Exercises

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

info Remove the output files after every exercise.

1) Split the `s1.txt` file 3 lines at a time.

add your solution here

```
$> head xa?
==> xaa <==
apple
coffee
fig

==> xab <==
honey
mango
pasta

==> xac <==
sugar
tea

$> rm xa?
```

2) Use appropriate options to get the output shown below.

```
$> echo 'apple,banana,cherry,dates' | ##### add your solution here
```

```
$> head xa?
==> xaa <==
apple,
==> xab <==
banana,
==> xac <==
cherry,
==> xad <==
dates

$> rm xa?
```

3) What do the -b and -C options do?

4) Display the 2nd chunk of the ip.txt file after splitting it 4 times as shown below.

```
##### add your solution here
come back before the sky turns dark
```

There are so many delights to cherish

5) What does the r prefix do when used with the -n option?

6) Split the ip.txt file 2 lines at a time. Customize the output filenames as shown below.

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

```
$> head ip_*
==> ip_0.txt <==
it is a warm and cozy day
listen to what I say

==> ip_1.txt <==
go play in the park
come back before the sky turns dark

==> ip_2.txt <==
```

There are so many delights to cherish

```
==> ip_3.txt <==
Apple, Banana and Cherry
Bread, Butter and Jelly
```

```
==> ip_4.txt <==
Try them all before you perish
```

```
$> rm ip_*
```

7) Which option would you use to prevent empty files in the output?

8) Split the items.txt file 5 lines at a time. Additionally, remove lines starting with a digit character as shown below.

```
$> cat items.txt
1) fruits
apple 5
banana 10
2) colors
green
sky blue
3) magical beasts
dragon 3
unicorn 42
```

add your solution here

```
$> head xa?
==> xaa <==
apple 5
banana 10
green
```

```
==> xab <==
sky blue
dragon 3
unicorn 42
```

```
$> rm xa?
```

csplit

The csplit command is useful to divide the input into smaller parts based on line numbers and regular expression patterns. Similar to split, this command also supports customizing output filenames.

info Since a lot of output files will be generated in this chapter (often with same filenames), remove these files after every illustration.

Split on Nth line

You can split the input into two based on a particular line number. To do so, specify the line number after the input source (filename or stdin data). The first output file will have the input lines before the given line number and the second output file will have the rest of the contents.

By default, the output files will be named xx00, xx01, xx02, and so on (where xx is the prefix). The numerical suffix will automatically use more digits if needed. You'll see examples with more than two output files later.

```
# split input into two based on line number 4
```

```
$> seq 10 | csplit - 4
```

```
6
15
```

```
# first output file will have the first 3 lines
```

```
# second output file will have the rest
```

```
$> head xx*
```

```
==> xx00 <==
```

```
1
2
3
```

```
==> xx01 <==
```

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

```
$> rm xx*
```

info As seen in the example above, csplit will also display the number of bytes written for each output file. You can use the -q option to suppress this message.

info warning As mentioned earlier, remove the output files after every illustration.

Split on regexp

You can also split the input based on a line matching the given regular expression. The output produced will vary based on the // or %% delimiters being used to surround the regexp.

When /regexp/ is used, output is similar to the line number based splitting. The first output file will have the input lines before the first occurrence of a line matching the given regexp and the second output file will have the rest of the contents.

```
# match a line containing 't' followed by zero or more characters and then 'p'
```

```
# 'toothpaste' is the only match for this input file
```

```
$> csplit -q purchases.txt '/t.*p/'
```

```
$> head xx*
```

```
==> xx00 <==
```

```
coffee
tea
washing powder
coffee
```

```
==> xx01 <==
```

```
toothpaste
tea
```

```
soap
tea
```

When %regexp% is used, the lines occurring before the matching line won't be part of the output. Only the line matching the given regexp and the rest of the contents will be part of the single output file.

```
$> csplit -q purchases.txt '%t.*p%'
```

```
$> cat xx00
toothpaste
tea
soap
tea
```

warning You'll get an error if the given regexp isn't found in the input.

```
$> csplit -q purchases.txt '/xyz/'
csplit: '/xyz/': match not found
```

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

Regexp offset

You can also provide offset numbers that'll affect where the matching line and its surrounding lines should be placed. When the offset is greater than zero, the split will happen that many lines after the matching line. The default offset is zero.

when the offset is '1', the matching line will be part of the first file

```
$> csplit -q purchases.txt '/t.*p/1'
```

```
$> head xx*
==> xx00 <==
coffee
tea
washing powder
coffee
toothpaste
```

```
==> xx01 <==
tea
soap
tea
```

matching line and 1 line after won't be part of the output

```
$> csplit -q purchases.txt '%t.*p%2'
```

```
$> cat xx00
soap
tea
```

When the offset is less than zero, the split will happen that many lines before the matching line.

2 lines before the matching line will be part of the second file

```
$> csplit -q purchases.txt '/t.*p/-2'
```

```
$> head xx*
==> xx00 <==
coffee
tea
```

```
==> xx01 <==
washing powder
coffee
toothpaste
tea
soap
tea
```

warning You'll get an error if the offset goes beyond the number of lines available in the input.

```
$> csplit -q purchases.txt '/t.*p/5'
csplit: '/t.*p/5': line number out of range
```

```
$> csplit -q purchases.txt '/t.*p/-5'
csplit: '/t.*p/-5': line number out of range
```

Repeat split

You can perform line number and regexp based split more than once by adding the {N} argument after the pattern. Default behavior examples seen so far is same as specifying {0}. Any number greater than zero will result in that many more splits.

```
# {1} means split one time more than the default split
# so, two splits in total and three output files
# in this example, split happens on the 4th and 8th line numbers
$> seq 10 | csplit -q - 4 '{1}'
```

```
$> head xx*
==> xx00 <==
1
```



```

2
3

==> xx01 <==
4
5
6
7

==> xx02 <==
8
9
10

```

Here's an example with regexp:

```

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

# split on the third (2+1) occurrence of a line containing 'warning'
$> csplit -q log.txt '%warning%' '{2}'
$> cat xx00
--> warning 3
4,3,1

```

As a special case, you can use `{*}` to repeat the split until the input is exhausted. This is especially useful with the `/regexp/` form of splitting. Here's an example:

```

# split on all lines matching 'paste' or 'powder'
$> csplit -q purchases.txt '/paste\|powder/' '{*}'
$> head xx*
==> xx00 <==
coffee
tea

==> xx01 <==
washing powder
coffee

==> xx02 <==
toothpaste
tea
soap
tea

```

warning You'll get an error if the repeat count goes beyond the number of matches possible with the given input.

```

$> seq 10 | csplit -q - 4 '{2}'
csplit: '4': line number out of range on repetition 2

```

```

$> csplit -q purchases.txt '/tea/' '{4}'
csplit: '/tea/': match not found on repetition 3

```

Keep files on error

By default, `csplit` will remove the created output files if there's an error or a signal that causes the command to stop. You can use the `-k` option to keep such files. One use case is line number based splitting with the `{*}` modifier.

```

$> seq 7 | csplit -q - 4 '{*}'
csplit: '4': line number out of range on repetition 1
$> ls xx*
ls: cannot access 'xx*': No such file or directory

```

`-k` option will allow you to retain the created files

```

$> seq 7 | csplit -qk - 4 '{*}'
csplit: '4': line number out of range on repetition 1
$> head xx*
==> xx00 <==
1
2
3

```

```

==> xx01 <==
4
5
6
7

```

Suppress matched lines

The `--suppress-matched` option will suppress the lines matching the split condition.

```
$> seq 5 | csplit -q --suppress-matched - 3
# 3rd line won't be part of the output
$> head xx*
==> xx00 <==
1
2

==> xx01 <==
4
5

$> rm xx*

$> seq 10 | csplit -q --suppress-matched - 4 '{1}'
# 4th and 8th lines won't be part of the output
$> head xx*
==> xx00 <==
1
2
3

==> xx01 <==
5
6
7

==> xx02 <==
9
10
```

Here's an example with regexp based split:

```
$> csplit -q --suppress-matched purchases.txt '/soap\|powder/' '{*}'
# lines matching 'soap' or 'powder' won't be part of the output
$> head xx*
==> xx00 <==
coffee
tea

==> xx01 <==
coffee
toothpaste
tea

==> xx02 <==
tea
```

Here's another example:

```
$> seq 11 16 | csplit -q --suppress-matched - '/[35]/' '{1}'
# lines matching '3' or '5' won't be part of the output
$> head xx*
==> xx00 <==
11
12

==> xx01 <==
14

==> xx02 <==
16

$> rm xx*
```

Exclude empty files

There are various cases that can result in empty output files. For example, first or last line matching the given split condition. Another possibility is the `--suppress-matched` option combined with consecutive lines matching during multiple splits. Here's an example:

```
$> csplit -q --suppress-matched purchases.txt '/coffee\|tea/' '{*}'

$> head xx*
==> xx00 <==

==> xx01 <==

==> xx02 <==
washing powder

==> xx03 <==
toothpaste

==> xx04 <==
soap
```

```
==> xx05 <==
```

You can use the `-z` option to exclude empty files from the output. The suffix numbering will be automatically adjusted in such cases.

```
$> csplit -qz --suppress-matched purchases.txt '/coffee\|tea/' '{*}'
```

```
$> head xx*
```

```
==> xx00 <==
```

```
washing powder
```

```
==> xx01 <==
```

```
toothpaste
```

```
==> xx02 <==
```

```
soap
```

Customize filenames

As seen earlier, `xx` is the default prefix for output filenames. Use the `-f` option to change this prefix.

```
$> seq 4 | csplit -q -f'num_' - 3
```

```
$> head num_*
```

```
==> num_00 <==
```

```
1
```

```
2
```

```
==> num_01 <==
```

```
3
```

```
4
```

The `-n` option controls the length of the numeric suffix. The suffix length will automatically increment if filenames are exhausted.

```
$> seq 4 | csplit -q -n1 - 3
```

```
$> ls xx*
```

```
xx00 xx01
```

```
$> rm xx*
```

```
$> seq 4 | csplit -q -n3 - 3
```

```
$> ls xx*
```

```
xx000 xx001
```

The `-b` option allows you to control the suffix using the `printf` formatting. Quoting from the manual: When this option is specified, the suffix string must include exactly one `printf(3)`-style conversion specification, possibly including format specification flags, a field width, a precision specifications, or all of these kinds of modifiers. The format letter must convert a binary unsigned integer argument to readable form. The format letters `d` and `i` are aliases for `u`, and the `u`, `o`, `x`, and `X` conversions are allowed.

Here are some examples:

```
# hexadecimal numbering
```

```
# minimum two digits, zero filled
```

```
$> seq 100 | csplit -q -b'%02x' - 3 '{20}'
```

```
$> ls xx*
```

```
xx00 xx02 xx04 xx06 xx08 xx0a xx0c xx0e xx10 xx12 xx14
```

```
xx01 xx03 xx05 xx07 xx09 xx0b xx0d xx0f xx11 xx13 xx15
```

```
$> rm xx*
```

```
# custom prefix and suffix around decimal numbering
```

```
# default minimum is a single digit
```

```
$> seq 20 | csplit -q -f'num_' -b'%d.txt' - 3 '{4}'
```

```
$> ls num_*
```

```
num_0.txt num_1.txt num_2.txt num_3.txt num_4.txt num_5.txt
```

info Note that the `-b` option will override the `-n` option. See `man 3 printf` for more details about the formatting options.

Exercises

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

info Remove the output files after every exercise.

1) Split the `blocks.txt` file such that the first 7 lines are in the first file and the rest are in the second file as shown below.

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

```
$> head xx*
```

```
==> xx00 <==
```

```
----
```

```
apple--banana
```

```
mango---fig
```

```
----
```

```
3.14
-42
1000
```

```
==> xx01 <==
----
sky blue
dark green
----
hi hello
```

```
$> rm xx*
```

2) Split the input file items.txt such that the text before a line containing colors is part of the first file and the rest are part of the second file as shown below.

add your solution here

```
$> head xx*
==> xx00 <==
1) fruits
apple 5
banana 10

==> xx01 <==
2) colors
green
sky blue
3) magical beasts
dragon 3
unicorn 42
```

```
$> rm xx*
```

3) Split the input file items.txt such that the line containing magical and all the lines that come after are part of the single output file.

add your solution here

```
$> cat xx00
3) magical beasts
dragon 3
unicorn 42
```

```
$> rm xx00
```

4) Split the input file items.txt such that the line containing colors as well the line that comes after are part of the first output file.

add your solution here

```
$> head xx*
==> xx00 <==
1) fruits
apple 5
banana 10
2) colors
green

==> xx01 <==
sky blue
3) magical beasts
dragon 3
unicorn 42
```

```
$> rm xx*
```

5) Split the input file items.txt on the line that comes before a line containing magical. Generate only a single output file as shown below.

add your solution here

```
$> cat xx00
sky blue
3) magical beasts
dragon 3
unicorn 42
```

```
$> rm xx00
```

6) Split the input file blocks.txt on the 4th occurrence of a line starting with the - character. Generate only a single output file as shown below.

add your solution here

```
$> cat xx00
```

```

----
sky blue
dark green
----
hi hello

```

```
$> rm xx00
```

7) For the input file blocks.txt, determine the logic to produce the expected output shown below.

add your solution here

```

$> head xx*
==> xx00 <==
apple--banana
mango---fig

```

```

==> xx01 <==
3.14
-42
1000

```

```

==> xx02 <==
sky blue
dark green

```

```

==> xx03 <==
hi hello

```

```
$> rm xx*
```

8) What does the -k option do?

9) Split the books.txt file on every line as shown below.

add your solution here

```
csplit: '1': line number out of range on repetition 3
```

```

$> head row_*
==> row_0 <==
Cradle::Mage Errant::The Weirkey Chronicles

```

```

==> row_1 <==
Mother of Learning::Eight::::Dear Spellbook:Ascendant

```

```

==> row_2 <==
Mark of the Fool:Super Powereds:::Ends of Magic

```

```
$> rm row_*
```

10) Split the items.txt file on lines starting with a digit character. Matching lines shouldn't be part of the output and the files should be named group_0.txt, group_1.txt and so on.

add your solution here

```

$> head group_*
==> group_0.txt <==
apple 5
banana 10

```

```

==> group_1.txt <==
green
sky blue

```

```

==> group_2.txt <==
dragon 3
unicorn 42

```

```
$> rm group_*
```

```

---
expand and unexpand

```

These two commands will help you convert tabs to spaces and vice versa. Both these commands support options to customize the width of tab stops and which occurrences should be converted.

Default expand

The expand command converts tab characters to space characters. The default expansion aligns at multiples of 8 columns (calculated in terms of bytes).

```

# sample stdin data
$> printf 'apple\tbanana\tcherry\na\tb\tc\n' | cat -T
apple^Ibanana^Icherry
a^Ib^Ic

```

```
cli-text-processing-with-gnu-coreutils-20250605.txt
```

```
# 'apple' = 5 bytes, \t converts to 3 spaces
# 'banana' = 6 bytes, \t converts to 2 spaces
# 'a' and 'b' = 1 byte, \t converts to 7 spaces
$> printf 'apple\tbanana\tcherry\n\tb\tc\n' | expand
apple  banana  cherry
a      b        c

# 'î±î»îµ' = 6 bytes, \t converts to 2 spaces
$> printf 'î±î»îµ\tî\200î¿î\215\n' | expand
î±î»îµ  î\200î¿î\215
```

Here's an example with strings of size 7 and 8 bytes before the tab character:

```
$> printf 'deviate\treached\nbackdrop\toverhang\n' | expand
deviate reached
backdrop          overhang
```

The expand command also considers backspace characters to determine the number of spaces needed.

```
# sample input with a backspace character
$> printf 'cart\bd\tbard\n' | cat -t
cart^Hd^Ibard

# 'card' = 4 bytes, \t converts to 4 spaces
$> printf 'cart\bd\tbard\n' | expand
card    bard
$> printf 'cart\bd\tbard\n' | expand | cat -t
cart^Hd    bard
```

info expand will concatenate multiple files passed as input source, so cat will not be needed for such cases.

Expand only the initial tabs

You can use the -i option to convert only the tab characters present at the start of a line. The first occurrence of a character that is not tab or space characters will stop the expansion.

```
# 'a' present at the start of line is not a tab/space character
# so no tabs are expanded for this input
$> printf 'a\tb\tc\n' | expand -i | cat -T
a^Ib^Ic
```

```
# the first \t gets expanded here, 'a' stops further expansion
$> printf '\ta\tb\tc\n' | expand -i | cat -T
a^Ib^Ic
```

```
# first two \t gets expanded here, 'a' stops further expansion
# presence of space characters will not stop the expansion
$> printf '\t \ta\tb\tc\n' | expand -i | cat -T
a^Ib^Ic
```

Customize the tab stop width

You can use the -t option to control the expansion width. Default is 8 as seen in the previous examples.

This option provides various features. Here's an example where all the tab characters are converted equally to the given width:

```
$> cat -T code.py
def compute(x, y):
^Iif x > y:
^I^Iprint('hello')
^Ielse:
^I^Iprint('bye')

$> expand -t 2 code.py
def compute(x, y):
  if x > y:
    print('hello')
  else:
    print('bye')
```

You can provide multiple widths separated by a comma character. In such a case, the given widths determine the stop locations for those many tab characters. These stop values refer to absolute positions from the start of the line, not the number of spaces they can expand to. Rest of the tab characters will be expanded to a single space character.

```
# first tab character can expand till the 3rd column
# second tab character can expand till the 7th column
# rest of the tab characters will be expanded to a single space
$> printf 'a\tb\tc\td\te\n' | expand -t 3,7
a  b   c d e

# here are two more examples with the same specification as above
# second tab expands to two spaces to end at the 7th column
$> printf 'a\tbb\tc\td\te\n' | expand -t 3,7
a  bb  c d e
# second tab expands to a single space since it goes beyond the 7th column
```

```
# 'card' = 4 bytes, so the 4 spaces gets converted to a tab
$> printf 'card\bd      bard\n' | unexpand -a | cat -T
card^Ibard
```

— — —

basename and dirname

These handy commands allow you to extract filenames and directory portions of the given paths. You could also use Parameter Expansion or cut, sed, awk, etc for such purposes. The advantage is that these commands will also handle corner cases like trailing slashes and there are handy features like removing file extensions.

Extract filename from paths

By default, the basename command will remove the leading directory component from the given path argument. Any trailing slashes will be removed before determining the portion to be extracted.

```
$> basename /home/learnbyexample/example_files/scores.csv
scores.csv
```

```
# quote the arguments when needed
$> basename 'path with spaces/report.log'
report.log
```

```
# one or more trailing slashes will not affect the output
$> basename /home/learnbyexample/example_files/
example_files
```

If there's no leading directory component or if slash alone is the input, the argument will be returned as is after removing any trailing slashes.

```
$> basename filename.txt
filename.txt
$> basename /
/
```

Remove file extension

You can use the -s option to remove a suffix from the filename. Usually used to remove the file extension.

```
$> basename -s'.csv' /home/learnbyexample/example_files/scores.csv
scores
```

```
$> basename -s'_2' final_report.txt_2
final_report.txt
```

```
$> basename -s'.tar.gz' /backups/jan_2021.tar.gz
jan_2021
```

```
$> basename -s'.txt' purchases.txt.txt
purchases.txt
```

```
# -s will be ignored if it would have resulted in an empty output
$> basename -s'report' /backups/report
report
```

You can also pass the suffix to be removed after the path argument, but the -s option is preferred as it makes the intention clearer and works for multiple path arguments.

```
$> basename example_files/scores.csv .csv
scores
```

Remove filename from path

By default, the dirname command removes the trailing path component (after removing any trailing slashes).

```
$> dirname /home/learnbyexample/example_files/scores.csv
/home/learnbyexample/example_files
```

```
# one or more trailing slashes will not affect the output
$> dirname /home/learnbyexample/example_files/
/home/learnbyexample
```

Multiple arguments

The dirname command accepts multiple path arguments by default. The basename command requires -a or -s (which implies -a) to work with multiple arguments.

```
$> basename -a /backups/jan_2021.tar.gz /home/learnbyexample/report.log
jan_2021.tar.gz
report.log
```

```
# -a is implied when the -s option is used
$> basename -s'.txt' logs/purchases.txt logs/report.txt
purchases
report
```

```
# dirname accepts multiple path arguments by default
```

```
$> dirname /home/learnbyexample/example_files/scores.csv ../report/backups/
/home/learnbyexample/example_files
../report
```

Combining basename and dirname

You can use shell features like command substitution to combine the effects of the basename and dirname commands.

```
# extract the second last path component
$> basename $(dirname /home/learnbyexample/example_files/scores.csv)
example_files

NUL separator
Use the -z option if you want to use NUL character as the output path separator.
```

```
$> basename -zs'.txt' logs/purchases.txt logs/report.txt | cat -v
purchases^@report^@
```

```
$> basename -z logs/purchases.txt | cat -v
purchases.txt^@
```

```
$> dirname -z example_files/scores.csv ../report/backups/ | cat -v
example_files^@../report^@
```

Exercises

1) Is the following command valid? If so, what would be the output?

```
$> basename -s.txt /test.txt//
```

2) Given the file path in the shell variable p, how'd you obtain the outputs shown below?

```
$> p='/projects/square_tictactoe/python/game.py'
##### add your solution here
/projects/square_tictactoe
```

```
$> p='/backups/jan_2021.tar.gz'
##### add your solution here
/
```

3) What would be the output of the basename command if the input has no leading directory component or only has the / character?

4) For the paths stored in the shell variable p, how'd you obtain the outputs shown below?

```
$> p='/a/b/ip.txt /c/d/e/f/op.txt'
```

```
# expected output 1
##### add your solution here
ip
op
```

```
# expected output 2
##### add your solution here
/a/b
/c/d/e/f
```

5) Given the file path in the shell variable p, how'd you obtain the outputs shown below?

```
$> p='/projects/python/square_tictactoe/game.py'
##### add your solution here
square_tictactoe
```

```
$> p='/backups/aug_2024/ip.tar.gz'
##### add your solution here
aug_2024
```
