Table of Contents

Use chat on gitter if you need help, have suggestions, etc

Command Line Text Processing

Work still in progress, stay tuned :)

Chapters

- Cat, Less, Tail and Head
  - cat, less, tail, head, Text Editors
- GNU grep
- GNU sed
- GNU awk
- Sorting stuff
  - sort, uniq, comm, shuf
- Restructure text
  - paste, column, pr, fold
- File attributes
  - wc, du, df, touch, file
- Miscellaneous
  - cut, tr, basename, dirname, xargs, seq

Webinar recordings

Am new to video recording and there are few bumps. But I hope it would be helpful

- Using the sort command
- Using uniq and comm

exercises

Check out exercises on github to test yourself, right from the command line itself

As of now, only `grep` exercises has been added. Stay tuned for more

Acknowledgements

- unix.stackexchange and stackoverflow - for getting answers to pertinent questions as well as sharpening skills by understanding and answering questions
- Forums like Linux users, /r/commandline/, /r/linux/, devup and others for valuable feedback (especially spotting mistakes) and encouragement

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Cat, Less, Tail and Head

Table of Contents

cat

```
$ man cat
CAT(1)                           User Commands                          CAT(1)

NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s) to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

- For below examples, `marks_201*` files contain 3 fields delimited by TAB
- To avoid formatting issues, TAB has been converted to spaces using `col -x` while pasting the output here

Concatenate files

- One or more files can be given as input and hence a lot of times, `cat` is used to quickly see contents of small single file on terminal
- To save the output of concatenation, just redirect stdout

```
$ ls
marks_2015.txt  marks_2016.txt  marks_2017.txt

$ cat marks_201*
Name    Maths   Science
foo     67      78
bar     87      85
Name    Maths   Science
foo     70      75
bar     85      88
Name    Maths   Science
foo     68      76
bar     90      90

$ # save stdout to a file
$ cat marks_201* > all_marks.txt
```

Accepting input from stdin

```
$ # combining input from stdin and other files
$ printf 'Name\tMaths\tScience \nbaz\t56\t63\nbak\t71\t65\n' | cat - marks_2015.txt
Name    Maths   Science
baz     56      63
bak     71      65
Name    Maths   Science
foo     67      78
bar     87      85

$ # - can be placed in whatever order is required
$ printf 'Name\tMaths\tScience \nbaz\t56\t63\nbak\t71\t65\n' | cat marks_2015.txt -
Name    Maths   Science
foo     67      78
bar     87      85
Name    Maths   Science
baz     56      63
bak     71      65
```

Squeeze consecutive empty lines

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


world

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

world

have a nice day
```

Prefix line numbers

```
$ # number all lines
$ cat -n marks_201*
     1  Name    Maths   Science
     2  foo     67      78
     3  bar     87      85
     4  Name    Maths   Science
     5  foo     70      75
     6  bar     85      88
     7  Name    Maths   Science
     8  foo     68      76
     9  bar     90      90

$ # number only non-empty lines
$ printf 'hello\n\n\nworld\n\nhave a nice day\n' | cat -sb
     1    hello

     2    world

     3    have a nice day
```

- For more numbering options, check out the command `nl`

```
$ whatis nl
nl (1)               - number lines of files
```

Viewing special characters

- End of line identified by `$`
- Useful for example to see trailing spaces

```
$ cat -E marks_2015.txt
Name    Maths   Science $
foo     67      78$
bar     87      85$
```

- TAB identified by `^I`

```
$ cat -T marks_2015.txt
Name^IMaths^IScience
foo^I67^I78
bar^I87^I85
```

- Non-printing characters
- See Show Non-Printing Characters for more detailed info

```
$ # NUL character
$ printf 'foo\0bar\0baz\n' | cat -v
foo^@bar^@baz

$ # to check for dos-style line endings
$ printf 'Hello World!\r\n' | cat -v
Hello World!^M

$ printf 'Hello World!\r\n' | dos2unix | cat -v
Hello World!
```

- the `-A` option is equivalent to `-vET`
- the `-e` option is equivalent to `-vE`
- If `dos2unix` and `unix2dos` are not available, see How to convert DOS/Windows newline (CRLF) to Unix newline (\n)

Writing text to file

```
$ cat > sample.txt
This is an example of adding text to a new file using cat command.
Press Ctrl+d on a newline to save and quit.

$ cat sample.txt
This is an example of adding text to a new file using cat command.
Press Ctrl+d on a newline to save and quit.
```

- See also how to use heredoc
  - How can I write a here doc to a file
- See also difference between Ctrl+c and Ctrl+d to signal end of stdin input in bash

tac

```
$ whatis tac
tac (1)              - concatenate and print files in reverse

$ seq 3 | tac
3
2
1

$ tac marks_2015.txt
bar     87     85
foo     67     78
Name     Maths     Science
```

- Useful in cases where logic is easier to write when working on reversed file
- Consider this made up log file, many **Warning** lines but need to extract only from last such **Warning** upto **Error** line

```
$ cat report.log
blah blah
Warning: something went wrong
more blah
whatever
Warning: something else went wrong
some text
some more text
Error: something seriously went wrong
blah blah blah

$ tac report.log | sed -n '/Error:/,/Warning:/p' | tac
Warning: something else went wrong
some text
some more text
Error: something seriously went wrong
```

- Similarly, if characters in lines have to be reversed, use the `rev` command

```
$ whatis rev
rev (1)              - reverse lines characterwise
```

Useless use of cat

- `cat` is used so frequently to view contents of a file that somehow users think other commands

cannot handle file input
- UUOC#Useless_use_of_cat)
- Useless Use of Cat Award

```
$ cat report.log | grep -E 'Warning|Error'
Warning: something went wrong
Warning: something else went wrong
Error: something seriously went wrong
$ grep -E 'Warning|Error' report.log
Warning: something went wrong
Warning: something else went wrong
Error: something seriously went wrong
```

- Use input redirection if a command doesn't accept file input

```
$ cat marks_2015.txt | tr 'A-Z' 'a-z'
name    maths   science
foo     67      78
bar     87      85
$ tr 'A-Z' 'a-z' < marks_2015.txt
name    maths   science
foo     67      78
bar     87      85
```

- However, `cat` should definitely be used where **concatenation** is needed

```
$ grep -c 'foo' marks_201*
marks_2015.txt:1
marks_2016.txt:1
marks_2017.txt:1

$ # concatenation allows to get overall count in one-shot in this case
$ cat marks_201* | grep -c 'foo'
3
```

Further Reading for cat

- cat Q&A on unix stackexchange
- cat Q&A on stackoverflow

less

```
$ whatis less
less (1)             - opposite of more

$ # By default, pager is used to display the man pages
$ # and usually, pager is linked to less command
$ type pager less
pager is /usr/bin/pager
less is /usr/bin/less

$ realpath /usr/bin/pager
/bin/less
$ realpath /usr/bin/less
/bin/less
$ diff -s /usr/bin/pager /usr/bin/less
Files /usr/bin/pager and /usr/bin/less are identical
```

- `cat` command is NOT suitable for viewing contents of large files on the Terminal
- `less` displays contents of a file, automatically fits to size of Terminal, allows scrolling in either direction and other options for effective viewing
- Usually, `man` command uses `less` command to display the help page
- The navigation commands are similar to `vi` editor

Navigation commands

Commonly used commands are given below, press `h` for summary of options

- `g` go to start of file
- `G` go to end of file
- `q` quit
- `/pattern` search for the given pattern in forward direction
- `?pattern` search for the given pattern in backward direction
- `n` go to next pattern
- `N` go to previous pattern

Further Reading for less

- See `man less` for detailed info on commands and options. For example:
    - `-s` option to squeeze consecutive blank lines
    - `-N` option to prefix line number

- `less` command is an improved version of `more` command
- differences between most, more and less
- less Q&A on unix stackexchange

tail

```
$ man tail
TAIL(1)                          User Commands                         TAIL(1)

NAME
       tail - output the last part of files

SYNOPSIS
       tail [OPTION]... [FILE]...

DESCRIPTION
       Print  the  last  10  lines of each FILE to standard output.  With more
       than one FILE, precede each with a header giving the file name.

       With no FILE, or when FILE is -, read standard input.
...
```

linewise tail

Consider this sample file, with line numbers prefixed

```
$ cat sample.txt
 1) Hello World!
 2)
 3) Good day
 4) How do you do?
 5)
 6) Just do it
 7) Believe it!
 8)
 9) Today is sunny
10) Not a bit funny
11) No doubt you like it too
12)
13) Much ado about nothing
14) He he he
15) Adios amigo
```

- default behavior - display last 10 lines

```
$ tail sample.txt
 6) Just do it
 7) Believe it!
 8)
 9) Today is sunny
10) Not a bit funny
11) No doubt you like it too
12)
13) Much ado about nothing
14) He he he
15) Adios amigo
```

- Use `-n` option to control number of lines to filter

```
$ tail -n3 sample.txt
13) Much ado about nothing
14) He he he
15) Adios amigo

$ # some versions of tail allow to skip explicit n character
$ tail -5 sample.txt
11) No doubt you like it too
12)
13) Much ado about nothing
14) He he he
15) Adios amigo
```

- when number is prefixed with `+` sign, all lines are fetched from that particular line number to end of file

```
$ tail -n +10 sample.txt
10) Not a bit funny
11) No doubt you like it too
12)
13) Much ado about nothing
14) He he he
15) Adios amigo

$ seq 13 17 | tail -n +3
15
16
17
```

characterwise tail

- Note that this works byte wise and not suitable for multi-byte character encodings

```
$ # last three characters including the newline character
$ echo 'Hi there!' | tail -c3
e!

$ # excluding the first character
$ echo 'Hi there!' | tail -c +2
i there!
```

multiple file input for tail

```
$ tail -n2 report.log sample.txt
==> report.log <==
Error: something seriously went wrong
blah blah blah

==> sample.txt <==
14) He he he
15) Adios amigo

$ # -q option to avoid filename in output
$ tail -q -n2 report.log sample.txt
Error: something seriously went wrong
blah blah blah
14) He he he
15) Adios amigo
```

Further Reading for tail

- `tail -f` and related options are beyond the scope of this tutorial. Below links might be useful
  - look out for buffering
  - Piping tail -f output though grep twice
  - tail and less
- tail Q&A on unix stackexchange
- tail Q&A on stackoverflow

head

```
$ man head
HEAD(1)                          User Commands                         HEAD(1)

NAME
       head - output the first part of files

SYNOPSIS
       head [OPTION]... [FILE]...

DESCRIPTION
       Print  the  first  10 lines of each FILE to standard output.  With more
       than one FILE, precede each with a header giving the file name.

       With no FILE, or when FILE is -, read standard input.
...
```

linewise head

- default behavior - display starting 10 lines

```
$ head sample.txt
 1) Hello World!
 2)
 3) Good day
 4) How do you do?
 5)
 6) Just do it
 7) Believe it!
 8)
 9) Today is sunny
10) Not a bit funny
```

- Use `-n` option to control number of lines to filter

```
$ head -n3 sample.txt
 1) Hello World!
 2)
 3) Good day

$ # some versions of head allow to skip explicit n character
$ head -4 sample.txt
 1) Hello World!
 2)
 3) Good day
 4) How do you do?
```

- when number is prefixed with `-` sign, all lines are fetched except those many lines to end of file

```
$ # except last 9 lines of file
$ head -n -9 sample.txt
 1) Hello World!
 2)
 3) Good day
 4) How do you do?
 5)
 6) Just do it

$ # except last 2 lines
$ seq 13 17 | head -n -2
13
14
15
```

characterwise head

- Note that this works byte wise and not suitable for multi-byte character encodings

```
$ # if output of command doesn't end with newline, prompt will be on same line
$ # to highlight working of command, the prompt for such cases is not shown here

$ # first two characters
$ echo 'Hi there!' | head -c2
Hi

$ # excluding last four characters
$ echo 'Hi there!' | head -c -4
Hi the
```

multiple file input for head

```
$ head -n3 report.log sample.txt
==> report.log <==
blah blah
Warning: something went wrong
more blah

==> sample.txt <==
 1) Hello World!
 2) 
 3) Good day

$ # -q option to avoid filename in output
$ head -q -n3 report.log sample.txt
blah blah
Warning: something went wrong
more blah
 1) Hello World!
 2) 
 3) Good day
```

combining head and tail

- Despite involving two commands, often this combination is faster than equivalent sed/awk versions

```
$ head -n11 sample.txt | tail -n3
 9) Today is sunny
10) Not a bit funny
11) No doubt you like it too

$ tail sample.txt | head -n2
 6) Just do it
 7) Believe it!
```

Further Reading for head

- head Q&A on unix stackexchange

Text Editors

For editing text files, the following applications can be used. Of these, `gedit` , `nano` , `vi` and/or `vim` are available in most distros by default

Easy to use

- gedit
- geany
- nano

Powerful text editors

- vim
    - vim learning resources and vim reference for further info
- emacs
- atom
- sublime

Check out this analysis for some performance/feature comparisons of various text editors

GNU grep

Table of Contents

```
$ grep -V | head -1
grep (GNU grep) 2.25

$ man grep
GREP(1)                      General Commands Manual                      GREP(1)

NAME
       grep, egrep, fgrep, rgrep - print lines matching a pattern

SYNOPSIS
       grep [OPTIONS] PATTERN [FILE...]
       grep [OPTIONS] [-e PATTERN]...  [-f FILE]...  [FILE...]

DESCRIPTION
       grep searches the named input FILEs for lines containing a match to the
       given PATTERN.  If no files are specified, or if the file “-” is given,
       grep  searches  standard  input.   By default, grep prints the matching
       lines.

       In addition, the variant programs egrep, fgrep and rgrep are  the  same
       as  grep -E,  grep -F,  and  grep -r, respectively.  These variants are
       deprecated, but are provided for backward compatibility.
...
```

Note For more detailed documentation and examples, use `info grep`

Simple string search

- First specify the search pattern (usually enclosed in single quotes) and then the file input
- More than one file can be specified or input given from stdin

```
$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ grep 'are' poem.txt
Roses are red,
Violets are blue,
And so are you.

$ grep 'so are' poem.txt
And so are you.
```

- If search string contains any regular expression meta characters like `^$\.*[]` (covered later), use the `-F` option or `fgrep` if available

```
$ echo 'int a[5]' | grep 'a[5]'
$ echo 'int a[5]' | grep -F 'a[5]'
int a[5]
$ echo 'int a[5]' | fgrep 'a[5]'
int a[5]
```

- See Gotchas and Tips section if you get strange issues

Case insensitive search

```
$ grep -i 'rose' poem.txt
Roses are red,

$ grep -i 'and' poem.txt
And so are you.
```

Invert matching lines

- Use the `-v` option to get lines other than those matching the search string
- Tip: Look out for other opposite pairs like `-l -L`, `-h -H`, opposites in regular expression, etc

```
$ grep -v 'are' poem.txt
Sugar is sweet,

$ # example for input from stdin
$ seq 5 | grep -v '3'
1
2
4
5
```

Line number, count and limiting output lines

- Show line number of matching lines

```
$ grep -n 'sweet' poem.txt
3:Sugar is sweet,
```

- Count number of matching lines

```
$ grep -c 'are' poem.txt
3
```

- Limit number of matching lines

```
$ grep -m2 'are' poem.txt
Roses are red,
Violets are blue,
```

Multiple search strings

- Match any

```
$ # search blue or you
$ grep -e 'blue' -e 'you' poem.txt
Violets are blue,
And so are you.
```

If there are lot of search strings, use a file input

```
$ printf 'rose\nsugar\n' > search_strings.txt
$ cat search_strings.txt
rose
sugar

$ # -f option accepts file input with search terms in separate lines
$ grep -if search_strings.txt poem.txt
Roses are red,
Sugar is sweet,
```

- Match all

```
$ # match line containing both are & And
$ grep 'are' poem.txt | grep 'And'
And so are you.
```

File names in output

- `-l` to get files matching the search
- `-L` to get files not matching the search
- `grep` skips the rest of file once a match is found

```
$ grep -l 'Rose' poem.txt
poem.txt

$ grep -L 'are' poem.txt search_strings.txt
search_strings.txt
```

- Prefix file name to search results
- `-h` is default for single file input, no file name prefix in output
- `-H` is default for multiple file input, file name prefix in output

```
$ grep -h 'Rose' poem.txt
Roses are red,
$ grep -H 'Rose' poem.txt
poem.txt:Roses are red,

$ # -H is default for multiple file input
$ grep -i 'sugar' poem.txt search_strings.txt
poem.txt:Sugar is sweet,
search_strings.txt:sugar
$ grep -ih 'sugar' poem.txt search_strings.txt
Sugar is sweet,
sugar
```

Match whole word or line

- Word search using `-w` option
  - word constitutes of alphabets, numbers and underscore character
- For example, this helps to distinguish `par` from `spar` , `part` , etc

```
$ printf 'par value\nheir apparent\n' | grep 'par'
par value
heir apparent

$ printf 'par value\nheir apparent\n' | grep -w 'par'
par value

$ printf 'scare\ncart\ncar\nmacaroni\n' | grep -w 'car'
car
```

- Another useful option is `-x` to match only complete line, not anywhere in the line

```
$ printf 'see my book list\nmy book\n' | grep 'my book'
see my book list
my book

$ printf 'see my book list\nmy book\n' | grep -x 'my book'
my book

$ printf 'scare\ncart\ncar\nmacaroni\n' | grep -x 'car'
car
```

Colored output

- Highlight search strings, line numbers, file name, etc in different colors
  - Depends on color support in terminal being used
- options to `--color` are
  - `auto` when output is redirected (another command, file, etc) the color information won't be passed
  - `always` when output is redirected (another command, file, etc) the color information will also be passed
  - `never` explicitly specify no highlighting

```
$ grep --color=auto 'blue' poem.txt
Violets are blue,
```

- Sample screenshot

- Example to show difference between `auto` and `always`

```
$ grep --color=auto 'blue' poem.txt > saved_output.txt
$ cat -v saved_output.txt
Violets are blue,
$ grep --color=always 'blue' poem.txt > saved_output.txt
$ cat -v saved_output.txt
Violets are ^[[01;31m^[[Kblue^[[m^[[K,
```

Get only matching portion

- The `-o` option to get only matched portion is more useful with regular expressions
- Comes in handy if overall number of matches is required, instead of only line wise

```
$ grep -o 'are' poem.txt
are
are
are

$ # -c only gives count of matching lines
$ grep -c 'e' poem.txt
4
$ grep -co 'e' poem.txt
4
$ # so need another command to get count of all matches
$ grep -o 'e' poem.txt | wc -l
9
```

Context matching

- The `-A` , `-B` and `-C` options are useful to get lines after/before/around matching line respectively

```
$ grep -A1 'blue' poem.txt
Violets are blue,
Sugar is sweet,
$ grep -B1 'blue' poem.txt
Roses are red,
Violets are blue,
$ grep -C1 'blue' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
```

- If there are multiple non-adjacent matching segments, by default `grep` adds a line `--` to separate them

```
$ seq 29 | grep -A1 '3'
3
4
--
13
14
--
23
24
```

- Use `--no-group-separator` option if the separator line is a hindrance, for example feeding the output of `grep` to another program

```
$ seq 29 | grep --no-group-separator -A1 '3'
3
4
13
14
23
24
```

- Use `--group-separator` to specify an alternate separator

```
$ seq 29 | grep --group-separator='*****' -A1 '3'
3
4
*****
13
14
*****
23
24
```

Recursive search

First let's create some more test files

```
$ mkdir -p test_files/hidden_files
$ printf 'Red\nGreen\nBlue\nBlack\nWhite\n' > test_files/colors.txt
$ printf 'Violet\nIndigo\nBlue\nGreen\nYellow\nOrange\nRed\n' > test_files/vibgyor.t
xt
$ printf '#!/usr/bin/python3\n\nprint("Hello World")\n' > test_files/hello.py
$ printf 'I like yellow\nWhat about you\n' > test_files/hidden_files/.fav_color.info
```

From `man grep`

```
       -r, --recursive
              Read all files  under  each  directory,  recursively,  following
              symbolic  links only if they are on the command line.  Note that
              if  no  file  operand  is  given,  grep  searches  the  working
              directory.  This is equivalent to the -d recurse option.

       -R, --dereference-recursive
              Read  all  files  under each directory, recursively.  Follow all
              symbolic links, unlike -r.
```

Basic recursive search

- Note that `-H` option automatically activates for multiple file input

```
$ # by default, current working directory is searched
$ grep -r 'red'
poem.txt:Roses are red,

$ grep -ri 'red'
poem.txt:Roses are red,
test_files/colors.txt:Red
test_files/vibgyor.txt:Red

$ grep -rin 'red'
poem.txt:1:Roses are red,
test_files/colors.txt:1:Red
test_files/vibgyor.txt:7:Red

$ grep -ril 'red'
poem.txt
test_files/colors.txt
test_files/vibgyor.txt
```

Exclude/Include specific files/directories

- By default, recursive search includes hidden files as well
- They can be excluded by file name or directory name
  - glob patterns can be used
  - for example: `*.[ch]` to specify all files ending with `.c` or `.h`
- The exclusion options can be used multiple times
  - for example: `--exclude='*.txt' --exclude='*.log'` or specified from a file using `--`

`exclude-from=FILE`

* To search only files with specific pattern in their names, use `--include=GLOB`
* **Note:** exclusion/inclusion applies only to basename of file/directory, not the entire path
* To follow all symbolic links (not directly specificied as arguments, but found on recursive search), use `-R` instead of `-r`

```
$ grep -ri 'you'
poem.txt:And so are you.
test_files/hidden_files/.fav_color.info:What about you

$ # exclude file names starting with `.` i.e hidden files
$ grep -ri --exclude='.*' 'you'
poem.txt:And so are you.

$ # include only file names ending with `.info`
$ grep -ri --include='*.info' 'you'
test_files/hidden_files/.fav_color.info:What about you

$ # exclude a directory
$ grep -ri --exclude-dir='hidden_files' 'you'
poem.txt:And so are you.

$ # If you are using git(or similar), this would be handy
$ # grep --exclude-dir='.git' -rl 'search pattern'
```

Recursive search with bash options

* Using `bash` options `globstar` (for recursion)
  * Other options like `extglob` and `dotglob` come in handy too
  * See glob for more info on these options
* The `-d skip` option tells grep to skip directories instead of trying to treat them as text file to be searched

```
$ grep -ril 'yellow'
test_files/hidden_files/.fav_color.info
test_files/vibgyor.txt

$ # recursive search
$ shopt -s globstar
$ grep -d skip -il 'yellow' **/*
test_files/vibgyor.txt

$ # include hidden files as well
$ shopt -s dotglob
$ grep -d skip -il 'yellow' **/*
test_files/hidden_files/.fav_color.info
test_files/vibgyor.txt

$ # use extended glob patterns
$ shopt -s extglob
$ # other than poem.txt
$ grep -d skip -il 'red' **/!(poem.txt)
test_files/colors.txt
test_files/vibgyor.txt
$ # other than poem.txt or colors.txt
$ grep -d skip -il 'red' **/!(poem|colors).txt
test_files/vibgyor.txt
```

Recursive search using find command

- `find` is obviously more versatile
- See also this guide for more examples/tutorials on using `find`

```
$ # all files, including hidden ones
$ find -type f -exec grep -il 'red' {} +
./poem.txt
./test_files/colors.txt
./test_files/vibgyor.txt

$ # all files ending with .txt
$ find -type f -name '*.txt' -exec grep -in 'you' {} +
./poem.txt:4:And so are you.

$ # all files not ending with .txt
$ find -type f -not -name '*.txt' -exec grep -in 'you' {} +
./test_files/hidden_files/.fav_color.info:2:What about you
```

Passing file names to other commands

- To pass files filtered to another command, see if the receiving command can differentiate file names by ASCII NUL character
- If so, use the `-Z` so that `grep` output is terminated with NUL character and commands like `xargs` have option `-0` to understand it
- This helps when file names can have characters like space, newline, etc
- Typical use case: Search and replace something in all files matching some pattern, for ex: `grep -rlZ 'PAT1' | xargs -0 sed -i 's/PAT2/REPLACE/g'`

```
$ # prompt at end of line not shown for simplicity
$ grep -rlZ 'you' | cat -A
poem.txt^@test_files/hidden_files/.fav_color.info^@

$ # print first column from all lines of all files
$ grep -rlZ 'you' | xargs -0 awk '{print $1}'
Roses
Violets
Sugar
And
I
What
```

- simple example to show filenames with space causing issue if `-Z` is not used

```
$ # 'abc xyz.txt' is a file with space in its name
$ grep -ri 'are'
abc xyz.txt:hi how are you
poem.txt:Roses are red,
poem.txt:Violets are blue,
poem.txt:And so are you.
saved_output.txt:Violets are blue,

$ # problem when -Z is not used
$ grep -ril 'are' | xargs grep 'you'
grep: abc: No such file or directory
grep: xyz.txt: No such file or directory
poem.txt:And so are you.

$ # no issues if -Z is used
$ grep -rilZ 'are' | xargs -0 grep 'you'
abc xyz.txt:hi how are you
poem.txt:And so are you.
```

- Example for matching more than one search string anywhere in file

```
$ # files containing 'you'
$ grep -rl 'you'
poem.txt
test_files/hidden_files/.fav_color.info

$ # files containing 'you' as well as 'are'
$ grep -rlZ 'you' | xargs -0 grep -l 'are'
poem.txt

$ # files containing 'you' but NOT 'are'
$ grep -rlZ 'you' | xargs -0 grep -L 'are'
test_files/hidden_files/.fav_color.info
```

- another example

```
$ grep -rilZ 'red' | xargs -0 grep -il 'blue'
poem.txt
test_files/colors.txt
test_files/vibgyor.txt

$ # note the use of `-Z` for middle command
$ grep -rilZ 'red' | xargs -0 grep -ilZ 'blue' | xargs -0 grep -il 'violet'
poem.txt
test_files/vibgyor.txt
```

Search strings from file

- using file input to specify search terms
- `-F` option will force matching strings literally(no regular expressions)
- See also Fastest way to find lines of a text file from another larger text file - read all answers

```
$ grep -if test_files/colors.txt poem.txt
Roses are red,
Violets are blue,

$ # get common lines between two files
$ grep -Fxf test_files/colors.txt test_files/vibgyor.txt
Blue
Green
Red

$ # get lines present in vibgyor.txt but not in colors.txt
$ grep -Fvxf test_files/colors.txt test_files/vibgyor.txt
Violet
Indigo
Yellow
Orange
```

Options for scripting purposes

- In scripts, often it is needed just to know if a pattern matches or not
- The `-q` option doesn't print anything on stdout and exit status is `0` if match is found
  - Check out this practical script using the `-q` option

```
$ grep -qi 'rose' poem.txt
$ echo $?
0
$ grep -qi 'lily' poem.txt
$ echo $?
1

$ if grep -qi 'rose' poem.txt; then echo 'match found!'; else echo 'match not found'
; fi
match found!
$ if grep -qi 'lily' poem.txt; then echo 'match found!'; else echo 'match not found'
; fi
match not found
```

- The `-s` option will suppress error messages as well

```
$ grep 'rose' file_xyz.txt
grep: file_xyz.txt: No such file or directory
$ grep -s 'rose' file_xyz.txt
$ echo $?
2

$ touch foo.txt
$ chmod -r foo.txt
$ grep 'rose' foo.txt
grep: foo.txt: Permission denied
$ grep -s 'rose' foo.txt
$ echo $?
2
```

Regular Expressions - BRE/ERE

Before diving into regular expressions, few examples to show default `grep` behavior vs `-F`

```
$ # oops, why did it not match?
$ echo 'int a[5]' | grep 'a[5]'

$ # where did that error come from??
$ echo 'int a[5]' | grep 'a['
grep: Invalid regular expression

$ # what is going on???
$ echo 'int a[5]' | grep 'a[5'
grep: Unmatched [ or [^

$ # phew, -F is a life saver
$ echo 'int a[5]' | grep -F 'a[5]'
int a[5]

$ # [ and ] are meta characters, details in following sections
$ echo 'int a[5]' | grep 'a\[5]'
int a[5]
```

- By default, `grep` treats the search pattern as BRE (Basic Regular Expression)
  - `-G` option can be used to specify explicitly that BRE is used
- The `-E` option allows to use ERE (Extended Regular Expression) which in GNU grep's case only differs in how meta characters are used, no difference in regular expression functionalities
- If `-F` option is used, the search string is treated literally

- If available, one can also use `-P` which indicates PCRE (Perl Compatible Regular Expression)

Line Anchors

- Often, search must match from beginning of line or towards end of line
- For example, an integer variable declaration in `C` will start with optional white-space, the keyword `int`, white-space and then variable(s)
    - This way one can avoid matching declarations inside single line comments as well.
- Similarly, one might want to match a variable at end of statement
- The meta characters for line anchoring are `^` for beginning of line and `$` for end of line

```
$ echo 'Fantasy is my favorite genre' > fav.txt
$ echo 'My favorite genre is Fantasy' >> fav.txt
$ cat fav.txt
Fantasy is my favorite genre
My favorite genre is Fantasy

$ # start of line
$ grep '^Fantasy' fav.txt
Fantasy is my favorite genre

$ # end of line
$ grep 'Fantasy$' fav.txt
My favorite genre is Fantasy

$ # without anchors
$ grep 'Fantasy' fav.txt
Fantasy is my favorite genre
My favorite genre is Fantasy
```

- As the meta characters have special meaning (assuming `-F` option is not used), they have to be escaped using `\` to match literally
- The `\` itself is meta character, so to match it literally, use `\\`
- The line anchors `^` and `$` have special meaning only when they are present at start/end of regular expression

```
$ echo '^foo bar$' | grep '^foo'
$ echo '^foo bar$' | grep '\^foo'
^foo bar$
$ echo '^foo bar$' | grep '^^foo'
^foo bar$

$ echo '^foo bar$' | grep 'bar$'
$ echo '^foo bar$' | grep 'bar\$'
^foo bar$
$ echo '^foo bar$' | grep 'bar$$'
^foo bar$

$ echo 'foo $ bar' | grep ' $ '
foo $ bar

$ printf 'foo\cbar' | grep -o '\c'
c
$ printf 'foo\cbar' | grep -o '\\c'
\c
```

Word Anchors

- The `-w` option works well to match whole words. But what about matching only start or end of words?
- Anchors `\<` and `\>` will match start/end positions of a word
- `\b` can also be used instead of `\<` and `\>` which matches either edge of a word

```
$ printf 'spar\npar\npart\napparent\n'
spar
par
part
apparent

$ # words ending with par
$ printf 'spar\npar\npart\napparent\n' | grep 'par\>'
spar
par

$ # words starting with par
$ printf 'spar\npar\npart\napparent\n' | grep '\<par'
par
part
```

- `-w` option is same as specifying both start and end word boundaries

```
$ printf 'spar\npar\npart\napparent\n' | grep '\<par\>'
par

$ printf 'spar\npar\npart\napparent\n' | grep '\bpar\b'
par

$ printf 'spar\npar\npart\napparent\n' | grep -w 'par'
par
```

- `\b` has an opposite `\B` which is quite useful too

```
$ # string not surrounded by word boundary either side
$ printf 'spar\npar\npart\napparent\n' | grep '\Bpar\B'
apparent

$ # word containing par but not as start of word
$ printf 'spar\npar\npart\napparent\n' | grep '\Bpar'
spar
apparent

$ # word containing par but not as end of word
$ printf 'spar\npar\npart\napparent\n' | grep 'par\B'
part
apparent
```

Alternation

- The `|` meta character is similar to using multiple `-e` option
- Each side of `|` is complete regular expression with their own start/end anchors
- How each part of alternation is handled and order of evaluation/output is beyond the scope of this tutorial
    - See this for more info on this topic.
- `|` is one of meta characters that requires different syntax between BRE/ERE

```
$ grep 'blue\|you' poem.txt
Violets are blue,
And so are you.
$ grep -E 'blue|you' poem.txt
Violets are blue,
And so are you.

$ # extract case-insensitive e or f from anywhere in line
$ echo 'Fantasy is my favorite genre' | grep -Eio 'e|f'
F
f
e
e
e

$ # extract case-insensitive e at end of line, f at start of line
$ echo 'Fantasy is my favorite genre' | grep -Eio 'e$|^f'
F
e
```

- A cool usecase of alternation is using `^` or `$` anchors to highlight searched term as well as display rest of unmatched lines
  - the line anchors will match every input line, even empty lines as they are position markers

```
$ grep --color=auto -E '^|are' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ grep --color=auto -E 'is|$' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

Screenshot for above example:

See also

- stackoverflow - Grep output with multiple Colors
- unix.stackexchange - Multicolored Grep

The dot meta character

The `.` meta character matches is used to match any character

```
$ # any two characters surrounded by word boundaries
$ echo 'I have 12, he has 132!' | grep -ow '..'
12
he

$ # match three characters from start of line
$ # \t (TAB) is single character here
$ printf 'a\tbcd\n' | grep -o '^...'
a	b

$ # all three character word starting with c
$ echo 'car bat cod cope scat dot abacus' | grep -ow 'c..'
car
cod

$ echo '1 & 2' | grep -o '.'
1
 
&
 
2
```

Quantifiers

Defines how many times a character (simplified for now) should be matched

- `?` will try to match 0 or 1 time
- For BRE, use `\?`

```
$ printf 'late\npale\nfactor\nrare\nact\n'
late
pale
factor
rare
act

$ # match a followed by t, with or without c in between
$ printf 'late\npale\nfactor\nrare\nact\n' | grep -E 'ac?t'
late
factor
act

$ # same as using this alternation
$ printf 'late\npale\nfactor\nrare\nact\n' | grep -E 'at|act'
late
factor
act
```

- `*` will try to match 0 or more times
- There is no upper limit and `*` will try to match as many times as possible

```
$ echo 'abbbc' | grep -o 'b*'
bbb

$ # matches 0 or more b only if surrounded by a and c
$ echo 'abc ac adc abbc bbb bc' | grep -o 'ab*c'
abc
ac
abbc

$ # see how it matched everything
$ echo 'car bat cod map scat dot abacus' | grep -o '.*'
car bat cod map scat dot abacus

$ # but here it stops at m
$ echo 'car bat cod map scat dot abacus' | grep -o '.*m'
car bat cod m

$ # stopped at dot, not bat or scat - match as much as possible
$ echo 'car bat cod map scat dot abacus' | grep -o 'c.*t'
car bat cod map scat dot

$ # matching overall expression gets preference
$ echo 'car bat cod map scat dot abacus' | grep -o 'c.*at'
car bat cod map scat

$ # precendence is left to right in case of multiple matches
$ echo 'car bat cod map scat dot abacus' | grep -o 'b.*m'
bat cod m
$ echo 'car bat cod map scat dot abacus' | grep -o 'b.*m*'
bat cod map scat dot abacus
```

- `+` will try to match 1 or more times
- Another meta character that differs in syntax between BRE/ERE

```
$ echo 'abbbc' | grep -o 'b\+'
bbb
$ echo 'abbbc' | grep -oE 'b+'
bbb

$ echo 'abc ac adc abbc bbb bc' | grep -oE 'ab+c'
abc
abbc
$ echo 'abc ac adc abbc bbb bc' | grep -o 'ab*c'
abc
ac
abbc
```

- For more precise control on number of times to match, `{}` (`\{\}` for BRE) is useful
- It can take one of four forms, `{n}` , `{n,m}` , `{,m}` and `{n,}`

```
$ # {n} - exactly n times
$ echo 'ac abc abbc abbbc' | grep -Eo 'ab{2}c'
abbc

$ # {n,m} - n to m, including both n and m
$ echo 'ac abc abbc abbbc' | grep -Eo 'ab{1,2}c'
abc
abbc

$ # {,m} - 0 to m times
$ echo 'ac abc abbc abbbc' | grep -Eo 'ab{,2}c'
ac
abc
abbc

$ # {n,} - at least n times
$ echo 'ac abc abbc abbbc' | grep -Eo 'ab{2,}c'
abbc
abbbc
```

Character classes

- The meta character pairs `[]` allow to match any of the multiple characters within `[]`
- Meta characters like `^` , `$` have different meaning inside and outside of `[]`
- Simple example first, matching any of the characters within `[]`

```
$ echo 'do so in to no on' | grep -ow '[nt]o'
to
no

$ echo 'do so in to no on' | grep -ow '[sot][on]'
so
to
on
```

- Adding a quantifier
- Check out unix words) and sample words file

```
$ # words made up of letters o and n, at least 2 letters
$ grep -xE '[on]{2,}' /usr/share/dict/words
no
non
noon
on

$ # lines containing only digits
$ printf 'cat\nfoo\n123\nbaz\n42\n' | grep -xE '[0123456789]+'
123
42
```

- Character ranges
- Matching any alphabet, number, hexadecimal number etc becomes cumbersome if every character has to be individually specified
- So, there's a shortcut, using `-` to construct a range (has to be specified in ascending order)
- See ascii codes table for reference
  - Note that behavior of range will differ for other character encodings
  - See **Character Classes and Bracket Expressions** as well as **LC_COLLATE under Environment Variables** sections in `info grep` for more detail
- Matching Numeric Ranges with a Regular Expression

```
$ printf 'cat\nfoo\n123\nbaz\n42\n' | grep -xE '[0-9]+'
123
42

$ printf 'cat\nfoo\n123\nbaz\n42\n' | grep -xiE '[a-z]+'
cat
foo
baz

$ # only valid decimal numbers
$ printf '128\n34\nfe32\nfoo1\nbar\n' | grep -xE '[0-9]+'
128
34

$ # only valid octal numbers
$ printf '128\n34\nfe32\nfoo1\nbar\n' | grep -xE '[0-7]+'
34

$ # only valid hexadecimal numbers
$ printf '128\n34\nfe32\nfoo1\nbar\n' | grep -xiE '[0-9a-f]+'
128
34
fe32

$ # numbers between 10-29
$ echo '23 54 12 92' | grep -owE '[12][0-9]'
23
12
```

- Negating character class
- By using `^` as first character inside `[]`, we get inverted character class
    - As pointed out earlier, some meta characters behave differently inside and outside of `[]`

```
$ # alphabetic words not starting with c
$ echo '123 core not sink code finish' | grep -owE '[^c][a-z]+'
not
sink
finish

$ # excluding numbers 2,3,4,9
$ # note that 200a 200; etc will also match, usage depends on knowing input
$ echo '2001 2004 2005 2008 2009' | grep -ow '200[^2-49]'
2001
2005
2008

$ # get characters from start of line upto(not including) known identifier
$ echo 'foo=bar; baz=123' | grep -oE '^[^=]+'
foo

$ # get characters at end of line from(not including) known identifier
$ echo 'foo=bar; baz=123' | grep -oE '[^=]+$'
123

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

- Matching meta characters inside `[]`
- Most meta characters like `( ) . + { } | $` don't have special meaning inside `[]` and hence do not require special treatment
- Some combination like `[.` or `=]` cannot be used in this order, as they have special meaning within `[]`
  - See **Character Classes and Bracket Expressions** section in `info grep` for more detail

```
$ # to match - it should be first or last character within []
$ echo 'Foo-bar 123-456 42 Co-operate' | grep -oiwE '[a-z-]+'
Foo-bar
Co-operate

$ # to match ] it should be first character within []
$ printf 'int a[5]\nfoo=bar\n' | grep '[]=]'
int a[5]
foo=bar

$ # to match [ use [ anywhere in the character list
$ # [][] will match both [ and ]
$ printf 'int a[5]\nfoo=bar\n' | grep '[[]'
int a[5]

$ # to match ^ it should be other than first in the list
$ echo '(a+b)^2 = a^2 + b^2 + 2ab' | grep -owE '[a-z^0-9]{3,}'
a^2
b^2
2ab
```

- Named character classes
- Equivalent class shown is for C locale and ASCII character encoding
  - See ascii codes table for reference
- See **Character Classes and Bracket Expressions** section in `info grep` for more detail

Character classes	Description
[:digit:]	Same as [0-9]
[:lower:]	Same as [a-z]
[:upper:]	Same as [A-Z]
[:alpha:]	Same as [a-zA-Z]
[:alnum:]	Same as [0-9a-zA-Z]
[:xdigit:]	Same as [0-9a-fA-F]
[:cntrl:]	Control characters - first 32 ASCII characters and 127th (DEL)
[:punct:]	All the punctuation characters
[:graph:]	[:alnum:] and [:punct:]
[:print:]	[:alnum:], [:punct:] and space
[:blank:]	Space and tab characters
[:space:]	white-space characters: tab, newline, vertical tab, form feed, carriage return and space

```
$ printf '128\n34\nAB32\nFoo\nbar\n' | grep -x '[[:alnum:]]*'
128
34
AB32
Foo
bar

$ printf '128\n34\nAB32\nFoo\nbar\n' | grep -x '[[:lower:]]*'
bar

$ printf '128\n34\nAB32\nFoo\nbar\n' | grep -x '[[:lower:]0-9]*'
128
34
bar
```

- backslash character classes
- The **word** `-w` option matches the same set of characters as that of `\w`

Character classes	Description
\w	Same as [0-9a-zA-Z*] or [[:alnum:]*]
\W	Same as 0-9a-zA-Z_ or [:alnum:]_
\s	Same as [[:space:]]
\S	Same as [:space:]

```
$ printf '123\n$#\ncmp_str\nFoo_bar\n' | grep -x '\w*'
123
cmp_str
Foo_bar
$ printf '123\n$#\ncmp_str\nFoo_bar\n' | grep -x '[[:alnum:]_]*'
123
cmp_str
Foo_bar

$ printf '123\n$#\ncmp_str\nFoo_bar\n' | grep -x '\W*'
$#
$ printf '123\n$#\ncmp_str\nFoo_bar\n' | grep -x '[^[:alnum:]_]*'
$#
```

Grouping

- Character classes allow matching against a choice of multiple character list and then quantifier added if needed

- One of the uses of grouping is analogous to character classes for whole regular expressions, instead of just list of characters
- The meta characters `()` are used for grouping
    - requires `\(\)` for BRE
- Similar to maths `ab + ac = a(b+c)`, think of regular expression `a(b|c) = ab|ac`

```
$ # 5 letter words starting with c and ending with ty or ly
$ grep -xE 'c..(ty|ly)' /usr/share/dict/words
catty
coyly
curly

$ # 7 letter words starting with e and ending with rged or sted
$ grep -xE 'e..(rg|st)ed' /usr/share/dict/words
emerged
existed

$ # repeat a pattern 3 times
$ grep -xE '([a-d][r-z]){3}' /usr/share/dict/words
avatar
awards
cravat

$ # nesting of () is allowed
$ grep -E '([as](p|c)[r-t]){2}' /usr/share/dict/words
scraps

$ # can be used to match specific columns in well defined tables
$ echo 'foo:123:bar:baz' | grep -E '^([^:]+:){2}bar'
foo:123:bar:baz
```

Back reference

- The matched string within `()` can also be used to be matched again by back referencing the captured groups
- `\1` denotes the first matched group, `\2` the second one and so on
    - Order is leftmost `(` is `\1`, next one is `\2` and so on
- Note that the matched string, not the regular expression itself is referenced
    - for ex: if `([0-9][a-f])` matches `3b`, then back referencing will be `3b` not any other valid match of the regular expression like `8f`, `0a` etc
    - Other regular expressions like PCRE do allow referencing the regular expression itself

```
$ # note how first three and last three letters are same
$ grep -xE '([a-d]..)\1' /usr/share/dict/words
bonbon
cancan
chichi
$ # note how adding quantifier is not same as back-referencing
$ grep -m4 -xE '([a-d]..){2}' /usr/share/dict/words
abacus
abided
abides
ablaze

$ # words with consecutive repeated letters
$ echo 'eel flee all pat ilk seen' | grep -iowE '[a-z]*(.)\1[a-z]*'
eel
flee
all
seen

$ # 17 letter words with first and last as same letter
$ grep -xE '(.)[a-z]{15}\1' /usr/share/dict/words
semiprofessionals
transcendentalist
```

- **Note** that there is an issue for certain usage of back-reference and quantifier

```
$ # no output
$ grep -m5 -xiE '([a-z]*([a-z])\2[a-z]*){2}' /usr/share/dict/words
$ # works when nesting is unrolled
$ grep -m5 -xiE '[a-z]*([a-z])\1[a-z]*([a-z])\2[a-z]*' /usr/share/dict/words
Abbott
Annabelle
Annette
Appaloosa
Appleseed

$ # no problem if PCRE is used instead of ERE
$ grep -m5 -xiP '([a-z]*([a-z])\2[a-z]*){2}' /usr/share/dict/words
Abbott
Annabelle
Annette
Appaloosa
Appleseed
```

- Useful to spot repeated words

- Use `-z` option (covered later) to match repetition in consecutive lines

```
$ cat story.txt
singing tin in the rain
walking for for a cause
have a nice day
day and night

$ grep -wE '(\w+)\W+\1' story.txt
walking for for a cause
```

Multiline matching

- If input is small enough to meet memory requirements, the `-z` option comes in handy to match across multiple lines
- Instead of newline being line separator, the ASCII NUL character is used
  - So, multiline matching depends on whether or not input file itself contains the NUL character
  - Usually text files won't have occasion to use the NUL character and presence of it marks it as binary file for `grep`

```
$ # \0 for ASCII NUL character
$ printf 'red\nblue\n\0green\n' | cat -e
red$
blue$
^@green$

$ # see --binary-files=TYPE option in info grep for binary details
$ printf 'red\nblue\n\0green\n' | grep -a 'red'
red

$ # with -z, \0 marks the different 'lines'
$ printf 'red\nblue\n\0green\n' | grep -z 'red'
red
blue

$ # if no \0 in input, entire input read as single string
$ printf 'red\nblue\ngreen\n' | grep -z 'red'
red
blue
green
```

- `\n` is not defined in BRE/ERE

  - see this for a workaround
- if some characteristics of input is known, `[[:space:]]` can be used as workaround, which matches all white-space characters

```
$ grep -oz 'Roses.*blue,[[:space:]]' poem.txt
Roses are red,
Violets are blue,
```

Perl Compatible Regular Expressions

```
$ # see also: https://github.com/learnbyexample/command_help
$ man grep | sed -n '/^\s*-P/,/^$/p'
       -P, --perl-regexp
              Interpret the pattern as a  Perl-compatible  regular  expression
              (PCRE).   This  is  highly  experimental and grep -P may warn of
              unimplemented features.
```

- The man page informs that `-P` is *highly experimental*. So far, haven't faced any issues. But do keep this in mind.
- Only a few highlights is presented here
- For more info
  - `man pcrepattern` or read it online
  - perldoc - re - Perl regular expression syntax, also links to other related tutorials
  - regular expression examples on SO documentation

Backslash sequences

Some of the backslash constructs available in PCRE over already seen ones in ERE

- `\d` for `[0-9]`
- `\s` for `[\ \t\r\n\f]`
- `\h` for `[ \t]`
- `\n` for newline character
- `\D` , `\S` , `\H` , `\N` etc for their opposites

```
$ # example for [0-9] in ERE and \d in PCRE
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oE '[0-9]+'
5
3
83
120
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oP '\d+'
5
3
83
120

$ # (?s) allows newlines to be also matches when using . meta character
$ grep -ozP '(?s)Roses.*blue,\n' poem.txt
Roses are red,
Violets are blue,
```

- See **INTERNAL OPTION SETTING** in `man pcrepattern` for more info on `(?s)` , `(?m)` etc
- Specifying Modes Inside The Regular Expression also has some detail on such options

Non-greedy matching

- Both BRE/ERE support only greedy matching quantifiers
  - match as much as possible
- PCRE supports non-greedy version by adding `?` after quantifiers
  - match as minimal as possible
- See this Python notebook for an interesting project on palindrome sentences

```
$ echo 'foo and bar and baz went shopping bytes' | grep -oi '\w.*and'
foo and bar and

$ echo 'foo and bar and baz went shopping bytes' | grep -oiP '\w.*?and'
foo and
bar and

$ # recall that matching overall expression gets preference
$ echo 'foo and bar and baz went shopping bytes' | grep -oi '\w.*and baz'
foo and bar and baz
$ echo 'foo and bar and baz went shopping bytes' | grep -oiP '\w.*?and baz'
foo and bar and baz

$ # minimal matching with single character has simple workaround
$ echo 'A man, a plan, a canal, Panama' | grep -oi 'a.*,'
A man, a plan, a canal,
$ echo 'A man, a plan, a canal, Panama' | grep -oi 'a[^,]*,'
A man,
a plan,
a canal,
```

Lookarounds

- Ability to add conditions to match before/after required pattern
- There are four types
  - positive lookahead `(?=`
  - negative lookahead `(?!`
  - positive lookbehind `(?<=`
  - negative lookbehind `(?<!`
- One way to remember is that **behind** uses `<` and **negative** uses `!` instead of `=`
- When used with `-o` option, lookarounds portion won't be part of output

Fixed and variable length *lookbehind*

```
$ # extract digits preceded by single lowercase letter and =
$ # this is fixed length lookbehind because length is known
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oP '(?<=\b[a-z]=)\d+'
83
120

$ # error because {2,} induces variable length matching
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oP '(?<=\b[a-z]{2,}=)\d+'
grep: lookbehind assertion is not fixed length

$ # use \K for such cases
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oP '\b[a-z]{2,}=\K\d+'
5
3
```

- Examples for lookarounds

```
$ # extract digits that follow =
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oP '=\K\d+'
5
3
83
120

$ # digits that follow = and has , after
$ echo 'foo=5, bar=3; x=83, y=120' | grep -oP '=\K\d+(?=,)'
5
83

$ # extract words, but not those at start of line
$ echo 'car bat cod map' | grep -owP '(?<!^)\w+'
bat
cod
map

$ # extract words, but not those at start of line or end of line
$ echo 'car bat cod map' | grep -owP '(?<!^)\w+(?!$)'
bat
cod
```

Ignoring specific matches

- A useful construct is `(*SKIP)(*F)` which allows to discard matches not needed

- Simple way to use is that regular expression which should be discarded is written first, `(*SKIP)(*F)` is appended and then whichever is required by added after `|`
- See Excluding Unwanted Matches for more info

```
$ # all words except bat and map
$ echo 'car bat cod map' | grep -oP '(bat|map)(*SKIP)(*F)|\w+'
car
cod

$ # all words except those surrounded by double quotes
$ echo 'I like "mango" and "guava"' | grep -oP '"[^"]+"(*SKIP)(*F)|\w+'
I
like
and
```

Re-using regular expression pattern

- `\1` , `\2` etc only matches exact string
- `(?1)` , `(?2)` etc re-uses the regular expression itself

```
$ # (?1) refers to first group \d{4}-\d{2}-\d{2}
$ echo '2008-03-24 and 2012-08-12 foo' | grep -oP '(\d{4}-\d{2}-\d{2})\D+(?1)'
2008-03-24 and 2012-08-12
```

Gotchas and Tips

- Always quote the search string (unless you know what you are doing :P)

```
$ grep so are poem.txt
grep: are: No such file or directory
poem.txt:And so are you.

$ grep 'so are' poem.txt
And so are you.
```

- Another common problem is unquoted search string will be open to shell's own globbing rules

```
$ # sample output on bash shell, might vary for different shells
$ echo '*.txt' | grep -F *.txt
$ echo '*.txt' | grep -F '*.txt'
*.txt
```

- Use double quotes for variable expansion, command substitution, etc (Note: could vary based on shell used)
- See mywiki.wooledge Quotes for detailed discussion of quoting in `bash` shell

```
$ # sample output on bash shell, might vary for different shells
$ color='blue'
$ grep "$color" poem.txt
Violets are blue,
```

- Pattern starting with `-`

```
$ # this issue is not specific to grep alone
$ # the command assumes -2 is an option and hence the error
$ echo '5*3-2=13' | grep '-2'
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.

$ # workaround by using \-
$ echo '5*3-2=13' | grep '\-2'
5*3-2=13

$ # or use -- to indicate no further options to process
$ echo '5*3-2=13' | grep -- '-2'
5*3-2=13

$ # same issue with printf
$ printf '-1+2=1\n'
bash: printf: -1: invalid option
printf: usage: printf [-v var] format [arguments]
$ printf -- '-1+2=1\n'
-1+2=1
```

- Tip: Options can be specified at end of command as well, useful if option was forgotten and have to quickly add it to previous command from history

```
$ grep 'are' poem.txt
Roses are red,
Violets are blue,
And so are you.

$ # use previous command from history, for ex up arrow key in bash
$ # then simply add the option at end
$ grep 'are' poem.txt -n
1:Roses are red,
2:Violets are blue,
4:And so are you.
```

- Speed boost if input file is ASCII

```
$ time grep -xE '([a-d][r-z]){3}' /usr/share/dict/words
avatar
awards
cravat

real    0m0.145s

$ time LC_ALL=C grep -xE '([a-d][r-z]){3}' /usr/share/dict/words
avatar
awards
cravat

real    0m0.011s
```

- Speed boost by using PCRE for back-references
- might be faster when using quantifiers as well

```
$ time LC_ALL=C grep -xE '([a-z]..)\1' /usr/share/dict/words
bonbon
cancan
chichi
murmur
muumuu
pawpaw
pompom
tartar
testes

real    0m0.174s
$ time grep -xP '([a-z]..)\1' /usr/share/dict/words
bonbon
cancan
chichi
murmur
muumuu
pawpaw
pompom
tartar
testes

real    0m0.008s
```

Regular Expressions Reference (ERE)

Anchors

- `^` match from start of line
- `$` match end of line
- `\<` match beginning of word
- `\>` match end of word
- `\b` match edge of word
- `\B` match other than edge of word

Character Quantifiers

- `.` match any single character

- `*` match preceding character/group 0 or more times
- `+` match preceding character/group 1 or more times
- `?` match preceding character/group 0 or 1 times
- `{n}` match preceding character/group exactly n times
- `{n,}` match preceding character/group n or more times
- `{n,m}` match preceding character/group n to m times, including n and m
- `{,m}` match preceding character/group up to m times

Character classes and backslash sequences

- `[aeiou]` match any of these characters
- `[^aeiou]` do not match any of these characters
- `[a-z]` match any lowercase alphabet
- `[0-9]` match any digit character
- `\w` match alphabets, digits and underscore character, short cut for `[a-zA-Z0-9_]`
- `\W` opposite of `\w` , short cut for `[^a-zA-Z0-9_]`
- `\s` match white-space characters: tab, newline, vertical tab, form feed, carriage return, and space
- `\S` match other than white-space characters

Pattern groups

- `|` matches either of the given patterns
- `()` patterns within `()` are grouped and treated as one pattern, useful in conjunction with `|`
- `\1` backreference to first grouped pattern within `()`
- `\2` backreference to second grouped pattern within `()` and so on

Basic vs Extended Regular Expressions

By default, the pattern passed to `grep` is treated as Basic Regular Expressions(BRE), which can be overridden using options like `-E` for ERE and `-P` for Perl Compatible Regular Expression(PCRE) Paraphrasing from `info grep`

> In Basic Regular Expressions the meta-characters `? + { | ( )` lose their special meaning, instead use the backslashed versions `\? \+ \{ \| \( \)`

Further Reading

- `man grep` and `info grep`
    - At least go through all options ;)
    - **Usage section** in `info grep` has good examples as well
- A bit of history
    - how grep command was born
    - why GNU grep is fast
    - Difference between grep, egrep and fgrep
- Tutorials and Q&A
    - grep tutorial
    - grep examples
    - grep Q&A on stackoverflow
    - grep Q&A on unix stackexchange
- Learn Regular Expressions (has information on flavors other than BRE/ERE/PCRE too)
    - Regular Expressions Tutorial
    - regexcrossword
    - What does this regex mean?
    - online regex tester and debugger - by default `pcre` flavor
- Alternatives
    - pcregrep
    - ag - silver searcher
    - ripgrep
- unix.stackexchange - When to use grep, sed, awk, perl, etc

GNU sed

Table of Contents

```
$ sed --version | head -n1
sed (GNU sed) 4.2.2

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

NAME
       sed - stream editor for filtering and transforming text

SYNOPSIS
       sed [OPTION]... {script-only-if-no-other-script} [input-file]...

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

Note: Multiline and manipulating pattern space with h,x,D,G,H,P etc is not covered in this chapter and examples/information is based on ASCII encoded text input only

Simple search and replace

Detailed examples for **substitute** command will be convered in later sections, syntax is

```
s/REGEXP/REPLACEMENT/FLAGS
```

The `/` character is idiomatically used as delimiter character. See also Using different delimiter for REGEXP

editing stdin

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

$ # change only first ',' to ' : '
$ seq 10 | paste -sd, | sed 's/,/ : /'
1 : 2,3,4,5,6,7,8,9,10

$ # change all ',' to ' : ' by using 'g' modifier
$ seq 10 | paste -sd, | sed 's/,/ : /g'
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10
```

Note: As a good practice, all examples use single quotes around arguments to prevent shell interpretation. See Shell substitutions section on use of double quotes

editing file input

- By default newline character is the line separator
- See Regular Expressions section for qualifying search terms
    - for example to distinguish between 'hi', 'this', 'his', 'history', etc

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

$ # change first 'Hi' in each line to 'Hello'
$ sed 's/Hi/Hello/' greeting.txt
Hello there
Have a nice day

$ # change first 'nice day' in each line to 'safe journey'
$ sed 's/nice day/safe journey/' greeting.txt
Hi there
Have a safe journey

$ # change all 'e' to 'E' and save changed text to another file
$ sed 's/e/E/g' greeting.txt > out.txt
$ cat out.txt
Hi thErE
HavE a nicE day
```

Inplace file editing

- In previous section, the output from `sed` was displayed on stdout or saved to another file
- To write the changes back to original file, use `-i` option

Note:

- Refer to `man sed` for details of how to use the `-i` option. It varies with different `sed` implementations. As mentioned at start of this chapter, `sed (GNU sed) 4.2.2` is being used here
- See this Q&A when working with symlinks

With backup

- When extension is given, the original input file is preserved with name changed according to extension provided

```
$ # '.bkp' is extension provided
$ sed -i.bkp 's/Hi/Hello/' greeting.txt

$ # original file gets preserved in 'greeting.txt.bkp'
Hi there
Have a nice day

$ # output from sed gets written to 'greeting.txt'
$ cat greeting.txt
Hello there
Have a nice day
```

Without backup

- Use this option with caution, changes made cannot be undone

```
$ sed -i 's/nice day/safe journey/' greeting.txt

$ # note, 'Hi' was already changed to 'Hello' in previous example
$ cat greeting.txt
Hello there
Have a safe journey
```

Multiple files

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

```
$ cat f1
I ate 3 apples
$ cat f2
I bought two bananas and 3 mangoes

$ # -i can be used with or without backup
$ sed -i 's/3/three/' f1 f2
$ cat f1
I ate three apples
$ cat f2
I bought two bananas and three mangoes
```

Prefix backup name

- A `*` in argument given to `-i` will get expanded to input filename
- This way, one can add prefix instead of suffix for backup

```
$ cat var.txt
foo
bar
baz

$ sed -i'bkp.*' 's/foo/hello/' var.txt
$ cat var.txt
hello
bar
baz

$ cat bkp.var.txt
foo
bar
baz
```

Place backups in directory

- `*` also allows to specify an existing directory to place the backups instead of current working directory

```
$ mkdir bkp_dir
$ sed -i'bkp_dir/*' 's/bar/hi/' var.txt
$ cat var.txt
hello
hi
baz

$ cat bkp_dir/var.txt
hello
bar
baz

$ # extensions can be added as well
$ # bkp_dir/*.bkp for suffix
$ # bkp_dir/bkp.* for prefix
$ # bkp_dir/bkp.*.2017 for both and so on
```

Line filtering options

- By default, `sed` acts on entire file. Often, one needs to extract or change only specific lines based on text search, line numbers, lines between two patterns, etc
- This filtering is much like using `grep` , `head` and `tail` commands in many ways and there are even more features
    - Use `sed` for inplace editing, the filtered lines to be transformed etc. Not as substitute for `grep` , `head` and `tail`

Print command

- It is usually used in conjunction with `-n` option
- By default, `sed` prints every input line, including any changes made by commands like substitution
    - printing here refers to line being part of `sed` output which may be shown on terminal, redirected to file, etc
- Using `-n` option and `p` command together, only specific lines needed can be filtered
- Examples below use the `/REGEXP/` addressing, other forms will be seen in sections to follow

```
$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ # all lines containing the string 'are'
$ # same as: grep 'are' poem.txt
$ sed -n '/are/p' poem.txt
Roses are red,
Violets are blue,
And so are you.

$ # all lines containing the string 'so are'
$ # same as: grep 'so are' poem.txt
$ sed -n '/so are/p' poem.txt
And so are you.
```

- Using print and substitution together

```
$ # print only lines on which substitution happens
$ sed -n 's/are/ARE/p' poem.txt
Roses ARE red,
Violets ARE blue,
And so ARE you.

$ # if line contains 'are', perform given command
$ # print only if substitution succeeds
$ sed -n '/are/ s/so/SO/p' poem.txt
And SO are you.
```

- Duplicating every input line

```
$ # note, -n is not used and no filtering applied
$ seq 3 | sed 'p'
1
1
2
2
3
3
```

Delete command

- By default, `sed` prints every input line, including any changes like substitution
- Using the `d` command, those specific lines will NOT be printed

```
$ # same as: grep -v 'are' poem.txt
$ sed '/are/d' poem.txt
Sugar is sweet,

$ # same as: seq 5 | grep -v '3'
$ seq 5 | sed '/3/d'
1
2
4
5
```

- Modifier `I` allows to filter lines in case-insensitive way
- See Regular Expressions section for more details

```
$ # /rose/I means match the string 'rose' irrespective of case
$ sed '/rose/Id' poem.txt
Violets are blue,
Sugar is sweet,
And so are you.
```

Quit commands

- Exit `sed` without processing further input

```
$ # same as: seq 23 45 | head -n5
$ # remember that printing is default action if -n is not used
$ seq 23 45 | sed '5q'
23
24
25
26
27
```

- `Q` is similar to `q` but won't print the matching line

```
$ seq 23 45 | sed '5Q'
23
24
25
26

$ # useful to print from beginning of file up to but not including line matching REG
EXP
$ sed '/is/Q' poem.txt
Roses are red,
Violets are blue,
```

- Use `tac` to get all lines starting from last occurrence of search string

```
$ # all lines from last occurrence of '7'
$ seq 50 | tac | sed '/7/q' | tac
47
48
49
50

$ # all lines from last occurrence of '7' excluding line with '7'
$ seq 50 | tac | sed '/7/Q' | tac
48
49
50
```

Note

- This way of using quit commands won't work for inplace editing with multiple file input
- See this Q&A for alternate solution as well using `gawk` and `perl` instead

Negating REGEXP address

- Use `!` to invert the specified address

```
$ # same as: sed -n '/so are/p' poem.txt
$ sed '/so are/!d' poem.txt
And so are you.

$ # same as: sed '/are/d' poem.txt
$ sed -n '/are/!p' poem.txt
Sugar is sweet,
```

Combining multiple REGEXP

- See also sed manual - Multiple commands syntax for more details
- See also sed scripts section to use a file for multiple commands

```
$ # each command as argument to -e option
$ sed -n -e '/blue/p' -e '/you/p' poem.txt
Violets are blue,
And so are you.

$ # each command separated by ;
$ # not all commands can be specified so
$ sed -n '/blue/p; /you/p' poem.txt
Violets are blue,
And so are you.

$ # each command separated by literal newline character
$ # might depend on whether the shell allows such multiline command
$ sed -n '
/blue/p
/you/p
' poem.txt
Violets are blue,
And so are you.
```

- Use `{}` command grouping for logical AND

```
$ # same as: grep 'are' poem.txt | grep 'And'
$ # space between /REGEXP/ and {} is optional
$ sed -n '/are/ {/And/p}' poem.txt
And so are you.

$ # same as: grep 'are' poem.txt | grep -v 'so'
$ sed -n '/are/ {/so/!p}' poem.txt
Roses are red,
Violets are blue,

$ # same as: grep -v 'red' poem.txt | grep -v 'blue'
$ sed -n '/red/!{/blue/!p}' poem.txt
Sugar is sweet,
And so are you.
$ # many ways to do it, use whatever feels easier to construct
$ # sed -e '/red/d' -e '/blue/d' poem.txt
$ # grep -v -e 'red' -e 'blue' poem.txt
```

- Different ways to do same things. See also Alternation and Control structures

```
$ # multiple commands can lead to duplicatation
$ sed -n '/blue/p; /t/p' poem.txt
Violets are blue,
Violets are blue,
Sugar is sweet,
$ # in such cases, use regular expressions instead
$ sed -nE '/blue|t/p;' poem.txt
Violets are blue,
Sugar is sweet,

$ sed -nE '/red|blue/!p' poem.txt
Sugar is sweet,
And so are you.

$ sed -n '/so/b; /are/p' poem.txt
Roses are red,
Violets are blue,
```

Filtering by line number

- Exact line number can be specified to be acted upon
- As a special case, `$` indicates last line of file
- See also sed manual - Multiple commands syntax

```
$ # here, 2 represents the address for print command, similar to /REGEXP/p
$ # same as: head -n2 poem.txt | tail -n1
$ sed -n '2p' poem.txt
Violets are blue,

$ # print 2nd and 4th line
$ # for `p`, `d`, `s` etc multiple commands can be specified separated by ;
$ sed -n '2p; 4p' poem.txt
Violets are blue,
And so are you.

$ # same as: tail -n1 poem.txt
$ sed -n '$p' poem.txt
And so are you.

$ # delete only 3rd line
$ sed '3d' poem.txt
Roses are red,
Violets are blue,
And so are you.
```

- For large input files, combine `p` with `q` for speedy exit
- `sed` would immediately quit without processing further input lines when `q` is used

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

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

$ # here is a sample time comparison
$ time seq 3542 4623452 | sed -n '2452{p;q}' > /dev/null

real    0m0.003s
user    0m0.000s
sys     0m0.000s
$ time seq 3542 4623452 | sed -n '2452p' > /dev/null

real    0m0.334s
user    0m0.396s
sys     0m0.024s
```

- mimicking `head` command using `q`

```
$ # same as: seq 23 45 | head -n5
$ # remember that printing is default action if -n is not used
$ seq 23 45 | sed '5q'
23
24
25
26
27
```

Print only line number

```
$ # gives both line number and matching line
$ grep -n 'blue' poem.txt
2:Violets are blue,

$ # gives only line number of matching line
$ sed -n '/blue/=' poem.txt
2

$ sed -n '/are/=' poem.txt
1
2
4
```

- If needed, matching line can also be printed. But there will be newline separation

```
$ sed -n '/blue/{=;p}' poem.txt
2
Violets are blue,

$ # or
$ sed -n '/blue/{p;=}' poem.txt
Violets are blue,
2
```

Address range

- So far, we've seen how to filter specific line based on *REGEXP* and line numbers
- `sed` also allows to combine them to enable selecting a range of lines
- Consider the sample input file for this section

```
$ cat addr_range.txt
Hello World

Good day
How are you

Just do-it
Believe it

Today is sunny
Not a bit funny
No doubt you like it too

Much ado about nothing
He he he
```

- Range defined by start and end *REGEXP*
- For other cases like getting lines without the line matching start and/or end, unbalanced start/end, when end *REGEXP* doesn't match, etc see Lines between two REGEXPs section

```
$ sed -n '/is/,/like/p' addr_range.txt
Today is sunny
Not a bit funny
No doubt you like it too

$ sed -n '/just/I,/believe/Ip' addr_range.txt
Just do-it
Believe it

$ # the second REGEXP will always be checked after the line matching first address
$ sed -n '/No/,/No/p' addr_range.txt
Not a bit funny
No doubt you like it too

$ # all the matching ranges will be printed
$ sed -n '/you/,/do/p' addr_range.txt
How are you

Just do-it
No doubt you like it too

Much ado about nothing
```

- Range defined by start and end line numbers

```
$ # print lines numbered 3 to 7
$ sed -n '3,7p' addr_range.txt
Good day
How are you

Just do-it
Believe it

$ # print lines from line number 13 to last line
$ sed -n '13,$p' addr_range.txt
Much ado about nothing
He he he

$ # delete lines numbered 2 to 13
$ sed '2,13d' addr_range.txt
Hello World
He he he
```

- Range defined by mix of line number and *REGEXP*

```
$ sed -n '3,/do/p' addr_range.txt
Good day
How are you

Just do-it

$ sed -n '/Today/,$p' addr_range.txt
Today is sunny
Not a bit funny
No doubt you like it too

Much ado about nothing
He he he
```

- Negating address range, just add `!` to end of address range

```
$ # same as: seq 10 | sed '3,7d'
$ seq 10 | sed -n '3,7!p'
1
2
8
9
10

$ # same as: sed '/Today/,$d' addr_range.txt
$ sed -n '/Today/,$!p' addr_range.txt
Hello World

Good day
How are you

Just do-it
Believe it
```

Relative addressing

- Prefixing `+` to a number for second address gives relative filtering
- Similar to using `grep -A<num> --no-group-separator 'REGEXP'` but `grep` merges adjacent groups while `sed` does not

```
$ # line matching 'is' and 2 lines after
$ sed -n '/is/,+2p' addr_range.txt
Today is sunny
Not a bit funny
No doubt you like it too

$ # note that all matching ranges will be filtered
$ sed -n '/do/,+2p' addr_range.txt
Just do-it
Believe it

No doubt you like it too

Much ado about nothing
```

- The first address could be number too
- Useful when using Shell substitutions

```
$ sed -n '3,+4p' addr_range.txt
Good day
How are you

Just do-it
Believe it
```

- Another relative format is `i~j` which acts on ith line and i+j, i+2j, i+3j, etc
    - `1~2` means 1st, 3rd, 5th, 7th, etc (i.e odd numbered lines)
    - `5~3` means 5th, 8th, 11th, etc

```
$ # match odd numbered lines
$ # for even, use 2~2
$ seq 10 | sed -n '1~2p'
1
3
5
7
9

$ # match line numbers: 2, 2+2*2, 2+3*2, etc
$ seq 10 | sed -n '2~4p'
2
6
10
```

- If `~j` is specified after `,` then meaning changes completely
- After the matching line based on number or *REGEXP* of start address, the closest line number multiple of `j` will mark end address

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

$ # line matching on `Just` is 6th line, so ending is 10th line
$ sed -n '/Just/,~5p' addr_range.txt
Just do-it
Believe it

Today is sunny
Not a bit funny
```

Using different delimiter for REGEXP

- `/` is idiomatically used as the *REGEXP* delimiter
  - See also a bit of history on why / is commonly used as delimiter
- But any character other than `\` and newline character can be used instead
- This helps to avoid/reduce use of `\`

```
$ # instead of this
$ echo '/home/learnbyexample/reports' | sed 's/\/home\/learnbyexample\//~\//'
~/reports

$ # use a different delimiter
$ echo '/home/learnbyexample/reports' | sed 's#/home/learnbyexample/#~/#'
~/reports
```

- For *REGEXP* used in address matching, syntax is a bit different `\<char>REGEXP<char>`

```
$ printf '/foo/bar/1\n/foo/baz/1\n'
/foo/bar/1
/foo/baz/1

$ printf '/foo/bar/1\n/foo/baz/1\n' | sed -n '\;/foo/bar/;p'
/foo/bar/1
```

Regular Expressions

- By default, `sed` treats *REGEXP* as BRE (Basic Regular Expression)
- The `-E` option enables ERE (Extended Regular Expression) which in GNU sed's case only differs in how meta characters are used, no difference in functionalities
    - Initially GNU sed only had `-r` option to enable ERE and `man sed` doesn't even mention `-E`
    - Other `sed` versions use `-E` and `grep` uses `-E` as well. So `-r` won't be used in examples in this tutorial
    - See also sed manual - BRE-vs-ERE
- See sed manual - Regular Expressions for more details

Line Anchors

- Often, search must match from beginning of line or towards end of line
- For example, an integer variable declaration in `C` will start with optional white-space, the keyword `int` , white-space and then variable(s)
    - This way one can avoid matching declarations inside single line comments as well
- Similarly, one might want to match a variable at end of statement

Consider the input file and sample substitution without using any anchoring

```
$ cat anchors.txt
cat and dog
too many cats around here
to concatenate, use the cmd cat
catapults laid waste to the village
just scat and quit bothering me
that is quite a fabricated tale
try the grape variety muscat

$ # without anchors, substitution will replace whereever the string is found
$ sed 's/cat/XXX/g' anchors.txt
XXX and dog
too many XXXs around here
to conXXXenate, use the cmd XXX
XXXapults laid waste to the village
just sXXX and quit bothering me
that is quite a fabriXXXed tale
try the grape variety musXXX
```

- The meta character `^` forces *REGEXP* to match only at start of line

```
$ # filtering lines starting with 'cat'
$ sed -n '/^cat/p' anchors.txt
cat and dog
catapults laid waste to the village

$ # replace only at start of line
$ # g modifier not needed as there can only be single match at start of line
$ sed 's/^cat/XXX/' anchors.txt
XXX and dog
too many cats around here
to concatenate, use the cmd cat
XXXapults laid waste to the village
just scat and quit bothering me
that is quite a fabricated tale
try the grape variety muscat

$ # add something to start of line
$ echo 'Have a good day' | sed 's/^/Hi! /'
Hi! Have a good day
```

- The meta character `$` forces *REGEXP* to match only at end of line

```
$ # filtering lines ending with 'cat'
$ sed -n '/cat$/p' anchors.txt
to concatenate, use the cmd cat
try the grape variety muscat

$ # replace only at end of line
$ sed 's/cat$/YYY/' anchors.txt
cat and dog
too many cats around here
to concatenate, use the cmd YYY
catapults laid waste to the village
just scat and quit bothering me
that is quite a fabricated tale
try the grape variety musYYY

$ # add something to end of line
$ echo 'Have a good day' | sed 's/$/. Cya later/'
Have a good day. Cya later
```

Word Anchors

- A **word** character is any alphabet (irrespective of case) or any digit or the underscore character
- The word anchors help in matching or not matching boundaries of a word
  - For example, to distinguish between `par` , `spar` and `apparent`
- `\b` matches word boundary
  - `\` is meta character and certain combinations like `\b` and `\B` have special meaning
- One can also use these alternatives for `\b`
  - `\<` for start of word
  - `\>` for end of word

```
$ # words ending with 'cat'
$ sed -n 's/cat\b/XXX/p' anchors.txt
XXX and dog
to concatenate, use the cmd XXX
just sXXX and quit bothering me
try the grape variety musXXX

$ # words starting with 'cat'
$ sed -n 's/\bcat/YYY/p' anchors.txt
YYY and dog
too many YYYs around here
to concatenate, use the cmd YYY
YYYapults laid waste to the village

$ # only whole words
$ sed -n 's/\bcat\b/ZZZ/p' anchors.txt
ZZZ and dog
to concatenate, use the cmd ZZZ

$ # word is made up of alphabets, numbers and _
$ echo 'foo, foo_bar and foo1' | sed 's/\bfoo\b/baz/g'
baz, foo_bar and foo1
```

- `\B` is opposite of `\b` , i.e it doesn't match word boundaries

```
$ # substitute only if 'cat' is surrounded by word characters
$ sed -n 's/\Bcat\B/QQQ/p' anchors.txt
to conQQQenate, use the cmd cat
that is quite a fabriQQQed tale

$ # substitute only if 'cat' is not start of word
$ sed -n 's/\Bcat/RRR/p' anchors.txt
to conRRRenate, use the cmd cat
just sRRR and quit bothering me
that is quite a fabriRRRed tale
try the grape variety musRRR

$ # substitute only if 'cat' is not end of word
$ sed -n 's/cat\B/SSS/p' anchors.txt
too many SSSs around here
to conSSSenate, use the cmd cat
SSSapults laid waste to the village
that is quite a fabriSSSed tale
```

Matching the meta characters

- Since meta characters like `^` , `$` , `\` etc have special meaning in *REGEXP*, they have to be escaped using `\` to match them literally

```
$ # here, '^' will match only start of line
$ echo '(a+b)^2 = a^2 + b^2 + 2ab' | sed 's/^/**/g'
**(a+b)^2 = a^2 + b^2 + 2ab

$ # '\` before '^' will match '^' literally
$ echo '(a+b)^2 = a^2 + b^2 + 2ab' | sed 's/\^/**/g'
(a+b)**2 = a**2 + b**2 + 2ab

$ # to match '\' use '\\'
$ echo 'foo\bar' | sed 's/\\/ /'
foo bar

$ echo 'pa$$' | sed 's/$/s/g'
pa$$s
$ echo 'pa$$' | sed 's/\$/s/g'
pass

$ # '^' has special meaning only at start of REGEXP
$ # similarly, '$' has special meaning only at end of REGEXP
$ echo '(a+b)^2 = a^2 + b^2 + 2ab' | sed 's/a^2/A^2/g'
(a+b)^2 = A^2 + b^2 + 2ab
```

- Certain characters like `&` and `\` have special meaning in *REPLACEMENT* section of substitute as well. They too have to be escaped using `\`
- And the delimiter character has to be escaped of course
- See back reference section for use of `&` in *REPLACEMENT* section

```
$ # & will refer to entire matched string of REGEXP section
$ echo 'foo and bar' | sed 's/and/"&"/'
foo "and" bar
$ echo 'foo and bar' | sed 's/and/"\&"/'
foo "&" bar

$ # use different delimiter where required
$ echo 'a b' | sed 's/ /\//'
a/b
$ echo 'a b' | sed 's# #/#'
a/b

$ # use \\ to represent literal \
$ echo '/foo/bar/baz' | sed 's#/#\\#g'
\foo\bar\baz
```

Alternation

- Two or more *REGEXP* can be combined as logical OR using the `|` meta character
  - syntax is `\|` for BRE and `|` for ERE
- Each side of `|` is complete regular expression with their own start/end anchors
- How each part of alternation is handled and order of evaluation/output is beyond the scope of this tutorial
  - See this for more info on this topic.

```
$ # BRE
$ sed -n '/red\|blue/p' poem.txt
Roses are red,
Violets are blue,

$ # ERE
$ sed -nE '/red|blue/p' poem.txt
Roses are red,
Violets are blue,

$ # filter lines starting or ending with 'cat'
$ sed -nE '/^cat|cat$/p' anchors.txt
cat and dog
to concatenate, use the cmd cat
catapults laid waste to the village
try the grape variety muscat

$ # g modifier is needed for more than one replacement
$ echo 'foo and temp and baz' | sed -E 's/foo|temp|baz/XYZ/'
XYZ and temp and baz
$ echo 'foo and temp and baz' | sed -E 's/foo|temp|baz/XYZ/g'
XYZ and XYZ and XYZ
```

The dot meta character

- The `.` meta character matches any character once, including newline

```
$ # replace all sequence of 3 characters starting with 'c' and ending with 't'
$ echo 'coat cut fit c#t' | sed 's/c.t/XYZ/g'
coat XYZ fit XYZ

$ # replace all sequence of 4 characters starting with 'c' and ending with 't'
$ echo 'coat cut fit c#t' | sed 's/c..t/ABCD/g'
ABCD cut fit c#t

$ # space, tab etc are also characters which will be matched by '.'
$ echo 'coat cut fit c#t' | sed 's/t.f/IJK/g'
coat cuIJKit c#t
```

Quantifiers

All quantifiers in `sed` are greedy, i.e longest match wins as long as overall *REGEXP* is satisfied and precedence is left to right. In this section, we'll cover usage of quantifiers on characters

- `?` will try to match 0 or 1 time
- For BRE, use `\?`

```
$ printf 'late\npale\nfactor\nrare\nact\n'
late
pale
factor
rare
act

$ # same as using: sed -nE '/at|act/p'
$ printf 'late\npale\nfactor\nrare\nact\n' | sed -nE '/ac?t/p'
late
factor
act

$ # greediness comes in handy in some cases
$ # problem: '<' has to be replaced with '\<' only if not preceded by '\'
$ echo 'blah \< foo bar < blah baz <'
blah \< foo bar < blah baz <
$ # this won't work as '\<' gets replaced with '\\<'
$ echo 'blah \< foo bar < blah baz <' | sed -E 's/</\\</g'
blah \\< foo bar \< blah baz \<
$ # by using '\\?<' both '\<' and '<' gets replaced by '\<'
$ echo 'blah \< foo bar < blah baz <' | sed -E 's/\\?</\\</g'
blah \< foo bar \< blah baz \<
```

- `*` will try to match 0 or more times

```
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n'
abc
ac
adc
abbc
bbb
bc
abbbbbc

$ # match 'a' and 'c' with any number of 'b' in between
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -n '/ab*c/p'
abc
ac
abbc
abbbbbc

$ # delete from start of line to 'te'
$ echo 'that is quite a fabricated tale' | sed 's/.*te//'
d tale
$ # delete from start of line to 'te '
$ echo 'that is quite a fabricated tale' | sed 's/.*te //'
a fabricated tale
$ # delete from first 'f' in the line to end of line
$ echo 'that is quite a fabricated tale' | sed 's/f.*//'
that is quite a
```

- `+` will try to match 1 or more times
- For BRE, use `\+`

```
$ # match 'a' and 'c' with at least one 'b' in between
$ # BRE
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -n '/ab\+c/p'
abc
abbc
abbbbbc

$ # ERE
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -nE '/ab+c/p'
abc
abbc
abbbbbc
```

- For more precise control on number of times to match, use `{}`

```
$ # exactly 5 times
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -nE '/ab{5}c/p'
abbbbbc

$ # between 1 to 3 times, inclusive of 1 and 3
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -nE '/ab{1,3}c/p'
abc
abbc

$ # maximum of 2 times, including 0 times
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -nE '/ab{,2}c/p'
abc
ac
abbc

$ # minimum of 2 times
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -nE '/ab{2,}c/p'
abbc
abbbbbc

$ # BRE
$ printf 'abc\nac\nadc\nabbc\nbbb\nbc\nabbbbbc\n' | sed -n '/ab\{2,\}c/p'
abbc
abbbbbc
```

Character classes

- The `.` meta character provides a way to match any character
- Character class provides a way to match any character among a specified set of characters enclosed within `[]`

```
$ # same as: sed -nE '/lane|late/p'
$ printf 'late\nlane\nfate\nfete\n' | sed -n '/la[nt]e/p'
late
lane

$ printf 'late\nlane\nfate\nfete\n' | sed -n '/[fl]a[nt]e/p'
late
lane
fate

$ # quantifiers can be added similar to using for any other character
$ # filter lines made up entirely of digits, containing at least one digit
$ printf 'cat5\nfoo\n123\n42\n' | sed -nE '/^[0123456789]+$/p'
123
42
$ # filter lines made up entirely of digits, containing at least three digits
$ printf 'cat5\nfoo\n123\n42\n' | sed -nE '/^[0123456789]{3,}$/p'
123
```

Character ranges

- Matching any alphabet, number, hexadecimal number etc becomes cumbersome if every character has to be individually specified
- So, there's a shortcut, using `-` to construct a range (has to be specified in ascending order)
- See ascii codes table for reference
    - Note that behavior of range will depend on locale settings
    - arch wiki - locale
    - Linux: Define Locale and Language Settings
- Matching Numeric Ranges with a Regular Expression

```
$ # filter lines made up entirely of digits, at least one
$ printf 'cat5\nfoo\n123\n42\n' | sed -nE '/^[0-9]+$/p'
123
42

$ # filter lines made up entirely of lower case alphabets, at least one
$ printf 'cat5\nfoo\n123\n42\n' | sed -nE '/^[a-z]+$/p'
foo

$ # filter lines made up entirely of lower case alphabets and digits, at least one
$ printf 'cat5\nfoo\n123\n42\n' | sed -nE '/^[a-z0-9]+$/p'
cat5
foo
123
42

$ # numbers between 10 to 29
$ printf '23\n154\n12\n26\n98234\n' | sed -n '/^[12][0-9]$/p'
23
12
26
$ # numbers >= 100
$ printf '23\n154\n12\n26\n98234\n' | sed -nE '/^[0-9]{3,}$/p'
154
98234
$ # numbers >= 100 if there are leading zeros
$ printf '0501\n035\n154\n12\n26\n98234\n' | sed -nE '/^0*[1-9][0-9]{2,}$/p'
0501
154
98234
```

Negating character class

- Meta characters inside and outside of `[]` are completely different
- For example, `^` as first character inside `[]` matches characters other than those specified inside character class

```
$ # delete all characters before first =
$ echo 'foo=bar; baz=123' | sed -E 's/^[^=]+//'
=bar; baz=123

$ # delete all characters after last =
$ echo 'foo=bar; baz=123' | sed -E 's/[^=]+$//'
foo=bar; baz=

$ # same as: sed -n '/[aeiou]/!p'
$ printf 'tryst\nglyph\npity\nwhy\n' | sed -n '/^[^aeiou]*$/p'
tryst
glyph
why
```

Matching meta characters inside `[]`

- Characters like `^` , `]` , `-` , etc need special attention to be part of list
- Also, sequences like `[.` or `=]` have special meaning within `[]`
    - See sed manual - Character-Classes-and-Bracket-Expressions for complete list

```
$ # to match - it should be first or last character within []
$ printf 'Foo-bar\n123-456\n42\nCo-operate\n' | sed -nE '/^[a-z-]+$/Ip'
Foo-bar
Co-operate

$ # to match ] it should be first character within []
$ printf 'int foo\nint a[5]\nfoo=bar\n' | sed -n '/[]=]/p'
int a[5]
foo=bar

$ # to match [ use [ anywhere in the character list
$ # [][] will match both [ and ]
$ printf 'int foo\nint a[5]\nfoo=bar\n' | sed -n '/[[]/p'
int a[5]

$ # to match ^ it should be other than first in the list
$ printf 'c=a^b\nd=f*h+e\nz=x-y\n' | sed -n '/[*^]/p'
c=a^b
d=f*h+e
```

Named character classes

- Equivalent class shown is for C locale and ASCII character encoding
    - See ascii codes table for reference
- See sed manual - Character Classes and Bracket Expressions for more details

Character classes	Description
[:digit:]	Same as [0-9]
[:lower:]	Same as [a-z]
[:upper:]	Same as [A-Z]
[:alpha:]	Same as [a-zA-Z]
[:alnum:]	Same as [0-9a-zA-Z]
[:xdigit:]	Same as [0-9a-fA-F]
[:cntrl:]	Control characters - first 32 ASCII characters and 127th (DEL)
[:punct:]	All the punctuation characters
[:graph:]	[:alnum:] and [:punct:]
[:print:]	[:alnum:], [:punct:] and space
[:blank:]	Space and tab characters
[:space:]	white-space characters: tab, newline, vertical tab, form feed, carriage return and space

```
$ # lines containing only hexadecimal characters
$ printf '128\n34\nfe32\nfoo1\nbar\n' | sed -nE '/^[[:xdigit:]]+$/p'
128
34
fe32

$ # lines containing at least one non-hexadecimal character
$ printf '128\n34\nfe32\nfoo1\nbar\n' | sed -n '/[^[:xdigit:]]/p'
foo1
bar

$ # same as: sed -nE '/^[a-z-]+$/Ip'
$ printf 'Foo-bar\n123-456\n42\nCo-operate\n' | sed -nE '/^[[:alpha:]-]+$/p'
Foo-bar
Co-operate

$ # remove all punctuation characters
$ sed 's/[[:punct:]]//g' poem.txt
Roses are red
Violets are blue
Sugar is sweet
And so are you
```

Backslash character classes

- Equivalent class shown is for C locale and ASCII character encoding
  - See ascii codes table for reference
- See sed manual - regular expression extensions for more details

Character classes	Description
\w	Same as [0-9a-zA-Z] or [[:alnum:]]
\W	Same as 0-9a-zA-Z_ or [:alnum:]_
\s	Same as [[:space:]]
\S	Same as [:space:]

```
$ # lines containing only word characters
$ printf '123\na=b+c\ncmp_str\nFoo_bar\n' | sed -nE '/^\w+$/p'
123
cmp_str
Foo_bar

$ # backslash character classes cannot be used inside [] unlike perl
$ # \w would simply match w
$ echo 'w=y-x+9*3' | sed 's/[\w=]//g'
y-x+9*3
$ echo 'w=y-x+9*3' | perl -pe 's/[\w=]//g'
-+*
```

Escape sequences

- Certain ASCII characters like tab, carriage return, newline, etc have escape sequence to represent them
  - Unlike backslash character classes, these can be used within `[]` as well
- Any ASCII character can be also represented using their decimal or octal or hexadecimal value
  - See ascii codes table for reference
- See sed manual - Escapes for more details

```
$ # example for representing tab character
$ printf 'foo\tbar\tbaz\n'
foo     bar     baz
$ printf 'foo\tbar\tbaz\n' | sed 's/\t/ /g'
foo bar baz
$ echo 'a b c' | sed 's/ /\t/g'
a       b       c

$ # using escape sequence inside character class
$ printf 'a\tb\vc\n'
a       b
         c
$ printf 'a\tb\vc\n' | cat -vT
a^Ib^Kc
$ printf 'a\tb\vc\n' | sed 's/[\t\v]/ /g'
a b c

$ # most common use case for hex escape sequence is to represent single quotes
$ # equivalent is '\d039' and '\o047' for decimal and octal respectively
$ echo "foo: '34'"
foo: '34'
$ echo "foo: '34'" | sed 's/\x27/"/g'
foo: "34"
$ echo 'foo: "34"' | sed 's/"/\x27/g'
foo: '34'
```

Grouping

- Character classes allow matching against a choice of multiple character list and then quantifier added if needed
- One of the uses of grouping is analogous to character classes for whole regular expressions, instead of just list of characters
- The meta characters `()` are used for grouping
  - requires `\(\)` for BRE
- Similar to maths `ab + ac = a(b+c)` , think of regular expression `a(b|c) = ab|ac`

```
$ # four letter words with 'on' or 'no' in middle
$ printf 'known\nmood\nknow\npony\ninns\n' | sed -nE '/\b[a-z](on|no)[a-z]\b/p'
know
pony
$ # common mistake to use character class, will match 'oo' and 'nn' as well
$ printf 'known\nmood\nknow\npony\ninns\n' | sed -nE '/\b[a-z][on]{2}[a-z]\b/p'
mood
know
pony
inns

$ # quantifier example
$ printf 'handed\nhand\nhandy\nhands\nhandle\n' | sed -nE '/^hand([sy]|le)?$/p'
hand
handy
hands
handle

$ # remove first two columns where : is delimiter
$ echo 'foo:123:bar:baz' | sed -E 's/^([^:]+:){2}//'
bar:baz

$ # can be nested as required
$ printf 'spade\nscore\nscare\nspare\nsphere\n' | sed -nE '/^s([cp](he|a)[rd])e$/p'
spade
scare
spare
sphere
```

Back reference

- The matched string within `()` can also be used to be matched again by back referencing the captured groups
- `\1` denotes the first matched group, `\2` the second one and so on
    - Order is leftmost `(` is `\1` , next one is `\2` and so on
    - Can be used both in *REGEXP* as well as in *REPLACEMENT* sections
- `&` or `\0` represents entire matched string in *REPLACEMENT* section
- Note that the matched string, not the regular expression itself is referenced
    - for ex: if `([0-9][a-f])` matches `3b` , then back referencing will be `3b` not any other valid match of the regular expression like `8f` , `0a` etc
- As `\` and `&` are special characters in *REPLACEMENT* section, use `\\` and `\&` respectively for literal representation

```
$ # filter lines with consecutive repeated alphabets
$ printf 'eel\nflee\nall\npat\nilk\nseen\n' | sed -nE '/([a-z])\1/p'
eel
flee
all
seen

$ # reduce \\ to single \ and delete if only single \
$ echo '\[\] and \\w and \[a-zA-Z0-9\_\]' | sed -E 's/(\\?)\\/\1/g'
[] and \w and [a-zA-Z0-9_]

$ # remove two or more duplicate words separated by space
$ # word boundaries prevent false matches like 'the theatre' 'sand and stone' etc
$ echo 'a a a walking for for a cause' | sed -E 's/\b(\w+)( \1)+\b/\1/g'
a walking for a cause

$ # surround only third column with double quotes
$ # note the nested capture groups and numbers used in REPLACEMENT section
$ echo 'foo:123:bar:baz' | sed -E 's/^(([^:]+:){2})([^:]+)/\1"\3"/'
foo:123:"bar":baz

$ # add first column data to end of line as well
$ echo 'foo:123:bar:baz' | sed -E 's/^([^:]+).*/& \1/'
foo:123:bar:baz foo

$ # surround entire line with double quotes
$ echo 'hello world' | sed 's/.*/"&"/'
"hello world"
$ # add something at start as well as end of line
$ echo 'hello world' | sed 's/.*/Hi. &. Have a nice day/'
Hi. hello world. Have a nice day
```

Changing case

- Applies only to *REPLACEMENT* section, unlike `perl` where these can be used in *REGEXP* portion as well
- See sed manual - The s Command for more details and corner cases

```
$ # UPPERCASE all alphabets, will be stopped on \L or \E
$ echo 'HeLlO WoRLD' | sed 's/.*/\U&/'
HELLO WORLD

$ # lowercase all alphabets, will be stopped on \U or \E
$ echo 'HeLlO WoRLD' | sed 's/.*/\L&/'
hello world

$ # Uppercase only next character
$ echo 'foo bar' | sed 's/\w*/\u&/g'
Foo Bar
$ echo 'foo_bar next_line' | sed -E 's/_([a-z])/\u\1/g'
fooBar nextLine

$ # lowercase only next character
$ echo 'FOO BAR' | sed 's/\w*/\l&/g'
fOO bAR
$ echo 'fooBar nextLine Baz' | sed -E 's/([a-z])([A-Z])/\1_\l\2/g'
foo_bar next_line Baz

$ # titlecase if input has mixed case
$ echo 'HeLlO WoRLD' | sed 's/.*/\L&/; s/\w*/\u&/g'
Hello World
$ # sed 's/.*/\L\u&/' also works, but not sure if it is defined behavior
$ echo 'HeLlO WoRLD' | sed 's/.*/\L&/; s/./\u&/'
Hello world

$ # \E will stop conversion started by \U or \L
$ echo 'foo_bar next_line baz' | sed -E 's/([a-z]+)(_[a-z]+)/\U\1\E\2/g'
FOO_bar NEXT_line baz
```

Substitute command modifiers

The `s` command syntax:

```
s/REGEXP/REPLACEMENT/FLAGS
```

- Modifiers (or FLAGS) like `g` , `p` and `I` have been already seen. For completeness, they will be discussed again along with rest of the modifiers
- See sed manual - The s Command for more details and corner cases

g modifier

By default, substitute command will replace only first occurrence of match. `g` modifier is needed to replace all occurrences

```
$ # replace only first : with -
$ echo 'foo:123:bar:baz' | sed 's/:/-/'
foo-123:bar:baz

$ # replace all : with -
$ echo 'foo:123:bar:baz' | sed 's/:/-/g'
foo-123-bar-baz
```

Replace specific occurrence

- A number can be used to specify *N*th match to be replaced

```
$ # replace first occurrence
$ echo 'foo:123:bar:baz' | sed 's/:/-/'
foo-123:bar:baz
$ echo 'foo:123:bar:baz' | sed -E 's/[^:]+/XYZ/'
XYZ:123:bar:baz

$ # replace second occurrence
$ echo 'foo:123:bar:baz' | sed 's/:/-/2'
foo:123-bar:baz
$ echo 'foo:123:bar:baz' | sed -E 's/[^:]+/XYZ/2'
foo:XYZ:bar:baz

$ # replace third occurrence
$ echo 'foo:123:bar:baz' | sed 's/:/-/3'
foo:123:bar-baz
$ echo 'foo:123:bar:baz' | sed -E 's/[^:]+/XYZ/3'
foo:123:XYZ:baz

$ # choice of quantifier depends on knowing input
$ echo ':123:bar:baz' | sed 's/[^:]*/XYZ/2'
:XYZ:bar:baz
$ echo ':123:bar:baz' | sed -E 's/[^:]+/XYZ/2'
:123:XYZ:baz
```

- Replacing *N*th match from end of line when number of matches is unknown
- Makes use of greediness of quantifiers

```
$ # replacing last occurrence
$ # can also use sed -E 's/:([^:]*)$/-\1/'
$ echo 'foo:123:bar:baz' | sed -E 's/(.*):/\1-/'
foo:123:bar-baz
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):/\1-/'
456:foo:123:bar:789-baz
$ echo 'foo and bar and baz and good' | sed -E 's/(.*)and/\1XYZ/'
foo and bar and baz XYZ good
$ # use word boundaries as necessary
$ echo 'foo and bar and baz land good' | sed -E 's/(.*)\band\b/\1XYZ/'
foo and bar XYZ baz land good

$ # replacing last but one
$ echo 'foo:123:bar:baz' | sed -E 's/(.*):(.*:)/\1-\2/'
foo:123-bar:baz
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):(.*:)/\1-\2/'
456:foo:123:bar-789:baz

$ # replacing last but two
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):((.*:){2})/\1-\2/'
456:foo:123-bar:789:baz
$ # replacing last but three
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):((.*:){3})/\1-\2/'
456:foo-123:bar:789:baz
```

- Replacing all but first *N* occurrences by combining with `g` modifier

```
$ # replace all : with - except first two
$ echo '456:foo:123:bar:789:baz' | sed -E 's/:/-/3g'
456:foo:123-bar-789-baz

$ # replace all : with - except first three
$ echo '456:foo:123:bar:789:baz' | sed -E 's/:/-/4g'
456:foo:123:bar-789-baz
```

- Replacing multiple *N*th occurrences

```
$ # replace first two occurrences of : with -
$ echo '456:foo:123:bar:789:baz' | sed 's/:/-/; s/:/-/'
456-foo-123:bar:789:baz

$ # replace second and third occurrences of : with -
$ # note the changes in number to be used for subsequent replacement
$ echo '456:foo:123:bar:789:baz' | sed 's/:/-/2; s/:/-/2'
456:foo-123-bar:789:baz

$ # better way is to use descending order
$ echo '456:foo:123:bar:789:baz' | sed 's/:/-/3; s/:/-/2'
456:foo-123-bar:789:baz
$ # replace second, third and fifth occurrences of : with -
$ echo '456:foo:123:bar:789:baz' | sed 's/:/-/5; s/:/-/3; s/:/-/2'
456:foo-123-bar:789-baz
```

Ignoring case

- Either `i` or `I` can be used for replacing in case-insensitive manner
- Since only `I` can be used for address filtering (for ex: `sed '/rose/Id' poem.txt`), use `I` for substitute command as well for consistency

```
$ echo 'hello Hello HELLO HeLlO' | sed 's/hello/hi/g'
hi Hello HELLO HeLlO

$ echo 'hello Hello HELLO HeLlO' | sed 's/hello/hi/Ig'
hi hi hi hi
```

p modifier

- Usually used in conjunction with `-n` option to output only modified lines

```
$ # no output if no substitution
$ echo 'hi there. have a nice day' | sed -n 's/xyz/XYZ/p'
$ # modified line if there is substitution
$ echo 'hi there. have a nice day' | sed -n 's/\bh/H/pg'
Hi there. Have a nice day

$ # only lines containing 'are'
$ sed -n 's/are/ARE/p' poem.txt
Roses ARE red,
Violets ARE blue,
And so ARE you.

$ # only lines containing 'are' as well as 'so'
$ sed -n '/are/ s/so/SO/p' poem.txt
And SO are you.
```

w modifier

- Allows to write only the changes to specified file name instead of default **stdout**

```
$ # space between w and filename is optional
$ # same as: sed -n 's/3/three/p' > 3.txt
$ seq 20 | sed -n 's/3/three/w 3.txt'
$ cat 3.txt
three
1three

$ # do not use -n if output should be displayed as well as written to file
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(:[^:]*){2}$//w col.txt'
456:foo:123:bar
$ cat col.txt
456:foo:123:bar
```

- For multiple output files, use `-e` for each file

```
$ seq 20 | sed -n -e 's/5/five/w 5.txt' -e 's/7/seven/w 7.txt'
$ cat 5.txt
five
1five
$ cat 7.txt
seven
1seven
```

- There are two predefined filenames
    - `/dev/stdout` to write to **stdout**
    - `/dev/stderr` to write to **stderr**

```
$ # inplace editing as well as display changes on terminal
$ sed -i 's/three/3/w /dev/stdout' 3.txt
3
13
$ cat 3.txt
3
13
```

e modifier

- Allows to use shell command output in *REPLACEMENT* section
- Trailing newline from command output is suppressed

```
$ # replacing a line with output of shell command
$ printf 'Date:\nreplace this line\n'
Date:
replace this line
$ printf 'Date:\nreplace this line\n' | sed 's/^replace.*/date/e'
Date:
Thu May 25 10:19:46 IST 2017

$ # when using p modifier with e, order is important
$ printf 'Date:\nreplace this line\n' | sed -n 's/^replace.*/date/ep'
Thu May 25 10:19:46 IST 2017
$ printf 'Date:\nreplace this line\n' | sed -n 's/^replace.*/date/pe'
date

$ # entire modified line is executed as shell command
$ echo 'xyz 5' | sed 's/xyz/seq/e'
1
2
3
4
5
```

m modifier

- Either `m` or `M` can be used
- So far, we've seen only line based operations (newline character being used to distinguish lines)
- There are various ways (see sed manual - How sed Works) by which more than one line is there in pattern space and in such cases `m` modifier can be used
- See also usage of multi-line modifier for more examples

Before seeing example with `m` modifier, let's see a simple example to get two lines in pattern space

```
$ # line matching 'blue' and next line in pattern space
$ sed -n '/blue/{N;p}' poem.txt
Violets are blue,
Sugar is sweet,

$ # applying substitution, remember that . matches newline as well
$ sed -n '/blue/{N;s/are.*is//p}' poem.txt
Violets  sweet,
```

- When `m` modifier is used, it affects the behavior of `^`, `$` and `.` meta characters

```
$ # without m modifier, ^ will anchor only beginning of entire pattern space
$ sed -n '/blue/{N;s/^/:: /pg}' poem.txt
:: Violets are blue,
Sugar is sweet,
$ # with m modifier, ^ will anchor each individual line within pattern space
$ sed -n '/blue/{N;s/^/:: /pgm}' poem.txt
:: Violets are blue,
:: Sugar is sweet,

$ # same applies to $ as well
$ sed -n '/blue/{N;s/$/ ::/pg}' poem.txt
Violets are blue,
Sugar is sweet, ::
$ sed -n '/blue/{N;s/$/ ::/pgm}' poem.txt
Violets are blue, ::
Sugar is sweet, ::

$ # with m modifier, . will not match newline character
$ sed -n '/blue/{N;s/are.*//p}' poem.txt
Violets
$ sed -n '/blue/{N;s/are.*//pm}' poem.txt
Violets
Sugar is sweet,
```

Shell substitutions

- Examples presented works with `bash` shell, might differ for other shells
- See also Difference between single and double quotes in Bash
- For robust substitutions taking care of meta characters in *REGEXP* and *REPLACEMENT* sections, see
    - How to ensure that string interpolated into sed substitution escapes all metachars
    - Is it possible to escape regex metacharacters reliably with sed

Variable substitution

- Entire command in double quotes can be used for simple use cases

```
$ word='are'
$ sed -n "/$word/p" poem.txt
Roses are red,
Violets are blue,
And so are you.

$ replace='ARE'
$ sed "s/$word/$replace/g" poem.txt
Roses ARE red,
Violets ARE blue,
Sugar is sweet,
And so ARE you.

$ # need to use delimiter as suitable
$ echo 'home path is:' | sed "s/$/ $HOME/"
sed: -e expression #1, char 7: unknown option to `s'
$ echo 'home path is:' | sed "s|$| $HOME|"
home path is: /home/learnbyexample
```

- If command has characters like `\`, backtick, `!` etc, double quote only the variable

```
$ # if history expansion is enabled, ! is special
$ word='are'
$ sed "/$word/!d" poem.txt
sed "/$word/date +%A" poem.txt
sed: -e expression #1, char 7: extra characters after command

$ # so double quote only the variable
$ # the command is concatenation of '/' and "$word" and '/!d'
$ sed '/'"$word"'/!d' poem.txt
Roses are red,
Violets are blue,
And so are you.
```

Command substitution

- Much more flexible than using `e` modifier as part of line can be modified as well

```
$ echo 'today is date' | sed 's/date/'"$(date +%A)"'/'
today is Tuesday

$ # need to use delimiter as suitable
$ echo 'current working dir is: ' | sed 's/$/'"$(pwd)"'/'
sed: -e expression #1, char 6: unknown option to `s'
$ echo 'current working dir is: ' | sed 's|$|'"$(pwd)"'|'
current working dir is: /home/learnbyexample/command_line_text_processing

$ # multiline output cannot be substituted in this manner
$ echo 'foo' | sed 's/foo/'"$(seq 5)"'/'
sed: -e expression #1, char 7: unterminated `s' command
```

z and s command line options

- We have already seen a few options like `-n` , `-e` , `-i` and `-E`
- This section will cover `-z` and `-s` options
- See sed manual - Command line options for other options and more details

The `-z` option will cause `sed` to separate input based on ASCII NUL character instead of newlines

```
$ # useful to process null separated data
$ # for ex: output of grep -Z, find -print0, etc
$ printf 'teal\0red\nblue\n\0green\n' | sed -nz '/red/p' | cat -A
red$
blue$
^@

$ # also useful to process whole file(not having NUL characters) as a single string
$ # adds ; to previous line if current line starts with c
$ printf 'cat\ndog\ncoat\ncut\nmat\n' | sed -z 's/\nc/;&/g'
cat
dog;
coat;
cut
mat
```

The `-s` option will cause `sed` to treat multiple input files separately instead of treating them as single concatenated input. If `-i` is being used, `-s` is implied

```
$ # without -s, there is only one first line
$ # F command prints file name of current file
$ sed '1F' f1 f2
f1
I ate three apples
I bought two bananas and three mangoes

$ # with -s, each file has its own address
$ sed -s '1F' f1 f2
f1
I ate three apples
f2
I bought two bananas and three mangoes
```

change command

The change command `c` will delete line(s) represented by address or address range and replace it with given string

Note the string used cannot have literal newline character, use escape sequence instead

```
$ # white-space between c and replacement string is ignored
$ seq 3 | sed '2c foo bar'
1
foo bar
3

$ # note how all lines in address range are replaced
$ seq 8 | sed '3,7cfoo bar'
1
2
foo bar
8

$ # escape sequences are allowed in string to be replaced
$ sed '/red/,/is/chello\nhi there' poem.txt
hello
hi there
And so are you.
```

- command will apply for all matching addresses

```
$ seq 5 | sed '/[24]/cfoo'
1
foo
3
foo
5
```

- `\` is special immediately after `c` , see sed manual - other commands for details
- If escape sequence is needed at beginning of replacement string, use an additional `\`

```
$ # \ helps to add leading spaces
$ seq 3 | sed '2c  a'
1
a
3
$ seq 3 | sed '2c\ a'
1
 a
3

$ seq 3 | sed '2c\tgood day'
1
tgood day
3
$ seq 3 | sed '2c\\tgood day'
1
        good day
3
```

- Since `;` cannot be used to distinguish between string and end of command, use `-e` for multiple commands

```
$ sed -e '/are/cHi;s/is/IS/' poem.txt
Hi;s/is/IS/
Hi;s/is/IS/
Sugar is sweet,
Hi;s/is/IS/

$ sed -e '/are/cHi' -e 's/is/IS/' poem.txt
Hi
Hi
Sugar IS sweet,
Hi
```

- Using shell substitution

```
$ text='good day'
$ seq 3 | sed '2c'"$text"
1
good day
3

$ text='good day\nfoo bar'
$ seq 3 | sed '2c'"$text"
1
good day
foo bar
3

$ seq 3 | sed '2c'"$(date +%A)"
1
Thursday
3

$ # multiline command output will lead to error
$ seq 3 | sed '2c'"$(seq 2)"
sed: -e expression #1, char 5: missing command
```

insert command

The insert command allows to add string before a line matching given address

Note the string used cannot have literal newline character, use escape sequence instead

```
$ # white-space between i and string is ignored
$ # same as: sed '2s/^/hello\n/'
$ seq 3 | sed '2i hello'
1
hello
2
3

$ # escape sequences can be used
$ seq 3 | sed '2ihello\nhi'
1
hello
hi
2
3
```

- command will apply for all matching addresses

```
$ seq 5 | sed '/[24]/ifoo'
1
foo
2
3
foo
4
5
```

- `\` is special immediately after `i`, see sed manual - other commands for details
- If escape sequence is needed at beginning of replacement string, use an additional `\`

```
$ seq 3 | sed '2i  foo'
1
foo
2
3
$ seq 3 | sed '2i\ foo'
1
 foo
2
3

$ seq 3 | sed '2i\tbar'
1
tbar
2
3
$ seq 3 | sed '2i\\tbar'
1
        bar
2
3
```

- Since `;` cannot be used to distinguish between string and end of command, use `-e` for multiple commands

```
$ sed -e '/is/ifoobar;s/are/ARE/' poem.txt
Roses are red,
Violets are blue,
foobar;s/are/ARE/
Sugar is sweet,
And so are you.

$ sed -e '/is/ifoobar' -e 's/are/ARE/' poem.txt
Roses ARE red,
Violets ARE blue,
foobar
Sugar is sweet,
And so ARE you.
```

- Using shell substitution

```
$ text='good day'
$ seq 3 | sed '2i'"$text"
1
good day
2
3

$ text='good day\nfoo bar'
$ seq 3 | sed '2i'"$text"
1
good day
foo bar
2
3

$ seq 3 | sed '2iToday is '"$(date +%A)"
1
Today is Thursday
2
3

$ # multiline command output will lead to error
$ seq 3 | sed '2i'"$(seq 2)"
sed: -e expression #1, char 5: missing command
```

append command

The append command allows to add string after a line matching given address

Note the string used cannot have literal newline character, use escape sequence instead

```
$ # white-space between a and string is ignored
$ # same as: sed '2s/$/\nhello/'
$ seq 3 | sed '2a hello'
1
2
hello
3

$ # escape sequences can be used
$ seq 3 | sed '2ahello\nhi'
1
2
hello
hi
3
```

- command will apply for all matching addresses

```
$ seq 5 | sed '/[24]/afoo'
1
2
foo
3
4
foo
5
```

- `\` is special immediately after `a` , see sed manual - other commands for details
- If escape sequence is needed at beginning of replacement string, use an additional `\`

```
$ seq 3 | sed '2a  foo'
1
2
foo
3
$ seq 3 | sed '2a\ foo'
1
2
 foo
3

$ seq 3 | sed '2a\tbar'
1
2
tbar
3
$ seq 3 | sed '2a\\tbar'
1
2
        bar
3
```

- Since `;` cannot be used to distinguish between string and end of command, use `-e` for multiple commands

```
$ sed -e '/is/afoobar;s/are/ARE/' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
foobar;s/are/ARE/
And so are you.

$ sed -e '/is/afoobar' -e 's/are/ARE/' poem.txt
Roses ARE red,
Violets ARE blue,
Sugar is sweet,
foobar
And so ARE you.
```

- Using shell substitution

```
$ text='good day'
$ seq 3 | sed '2a'"$text"
1
2
good day
3

$ text='good day\nfoo bar'
$ seq 3 | sed '2a'"$text"
1
2
good day
foo bar
3

$ seq 3 | sed '2aToday is '"$(date +%A)"
1
2
Today is Thursday
3

$ # multiline command output will lead to error
$ seq 3 | sed '2a'"$(seq 2)"
sed: -e expression #1, char 5: missing command
```

- See this Q&A for using `a` command to make sure last line of input has a newline character

adding contents of file

r for entire file

- The `r` command allows to add contents of file after a line matching given address
- It is a robust way to add multiline content or if content can have characters that may be interpreted
- Special name `/dev/stdin` allows to read from **stdin** instead of file input
- First, a simple example to add contents of one file into another at specified address

```
$ cat 5.txt
five
1five

$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ # space between r and filename is optional
$ sed '2r 5.txt' poem.txt
Roses are red,
Violets are blue,
five
1five
Sugar is sweet,
And so are you.

$ # content cannot be added before first line
$ sed '0r 5.txt' poem.txt
sed: -e expression #1, char 2: invalid usage of line address 0
$ # but that is trivial to solve: cat 5.txt poem.txt
```

- command will apply for all matching addresses

```
$ seq 5 | sed '/[24]/r 5.txt'
1
2
five
1five
3
4
five
1five
5
```

- adding content of variable as it is without any interpretation
- also shows example for using `/dev/stdin`

```
$ text='Good day\nfoo bar baz\n'
$ # escape sequence like \n will be interpreted when 'a' command is used
$ sed '/is/a'"$text" poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
Good day
foo bar baz

And so are you.

$ # \ is just another character, won't be treated as special with 'r' command
$ echo "$text" | sed '/is/r /dev/stdin' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
Good day\nfoo bar baz\n
And so are you.
```

- adding multiline command output is simple as well

```
$ seq 3 | sed '/is/r /dev/stdin' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
1
2
3
And so are you.
```

- replacing a line or range of lines with contents of file

```
$ # replacing range of lines
$ # order is important, first 'r' and then 'd'
$ sed -e '/is/r 5.txt' -e '1,/is/d' poem.txt
five
1five
And so are you.

$ # replacing a line
$ seq 3 | sed -e '3r /dev/stdin' -e '3d' poem.txt
Roses are red,
Violets are blue,
1
2
3
And so are you.

$ # can also use {} grouping
$ seq 3 | sed -e '/blue/{r /dev/stdin' -e 'd}' poem.txt
Roses are red,
1
2
3
Sugar is sweet,
And so are you.
```

R for line by line

- add a line for every address match
- Special name `/dev/stdin` allows to read from **stdin** instead of file input

```
$ # space between R and filename is optional
$ seq 3 | sed '/are/R /dev/stdin' poem.txt
Roses are red,
1
Violets are blue,
2
Sugar is sweet,
And so are you.
3

$ sed '2,3R 5.txt' poem.txt
Roses are red,
Violets are blue,
five
Sugar is sweet,
1five
And so are you.
```

- number of lines from file to be read different from number of matching address lines

```
$ # file has more lines than matching address
$ sed '/is/R 5.txt' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
five
And so are you.

$ # lines matching address is more than file to be read
$ seq 1 | sed '/are/R /dev/stdin' poem.txt
Roses are red,
1
Violets are blue,
Sugar is sweet,
And so are you.
```

n and N commands

- These two commands will fetch next line (newline or NUL character separated, depending on options)
- `n` will fetch the next line and replace whatever is already there in pattern space

```
$ # if line contains 'blue', replace 'e' with 'E' only for following line
$ sed '/blue/{n;s/e/E/g}' poem.txt
Roses are red,
Violets are blue,
Sugar is swEEt,
And so are you.

$ # better illustrated with -n option
$ sed -n '/blue/{n;s/e/E/pg}' poem.txt
Sugar is swEEt,

$ # if line contains 'blue', replace 'e' with 'E' only for next to next line
$ sed -n '/blue/{n;n;s/e/E/pg}' poem.txt
And so arE you.
```

- `N` will fetch the next line and append to pattern space
- See this Q&A for an interesting case of applying substitution every 4 lines but excluding the 4th line

```
$ # if line contains 'blue', replace 'e' with 'E' both in current line and next
$ sed '/blue/{N;s/e/E/g}' poem.txt
Roses are red,
ViolEts arE bluE,
Sugar is swEEt,
And so are you.

$ # better illustrated with -n option
$ sed -n '/blue/{N;s/e/E/pg}' poem.txt
ViolEts arE bluE,
Sugar is swEEt,

$ sed -n '/blue/{N;N;s/e/E/pg}' poem.txt
ViolEts arE bluE,
Sugar is swEEt,
And so arE you.
```

- Combination

```
$ # n will fetch next line, current line is out of pattern space
$ # N will then add another line
$ sed -n '/blue/{n;N;s/e/E/pg}' poem.txt
Sugar is swEEt,
And so arE you.
```

- not necessary to qualify with an address

```
$ seq 6 | sed 'n;cXYZ'
1
XYZ
3
XYZ
5
XYZ

$ seq 6 | sed 'N;s/\n/ /'
1 2
3 4
5 6
```

Control structures

- Using `:label` one can mark a command location to branch to conditionally or unconditionally
- See sed manual - Commands for sed gurus for more details

if then else

- Simple if-then-else can be simulated using `b` command
- `b` command will unconditionally branch to specified label
- Without label, `b` will skip rest of commands and start next cycle
- See processing only lines between REGEXPs for interesting use case

```
$ # changing -ve to +ve and vice versa
$ cat nums.txt
42
-2
10101
-3.14
-75
$ # same as: perl -pe '/^-/ ? s/// : s/^/-/'
$ # empty REGEXP section will reuse previous REGEXP, in this case /^-/
$ sed '/^-/{s///;b}; s/^/-/' nums.txt
-42
2
-10101
3.14
75

$ # same as: perl -pe '/are/ ? s/e/*/g : s/e/#/g'
$ # if line contains 'are' replace 'e' with '*' else replace 'e' with '#'
$ sed '/are/{s/e/*/g;b}; s/e/#/g' poem.txt
Ros*s ar* r*d,
Viol*ts ar* blu*,
Sugar is sw##t,
And so ar* you.
```

replacing in specific column

- `t` command will branch to specified label on successful substitution
- Without label, `t` will skip rest of commands and start next cycle
- More examples
  - replace data after last delimiter
  - replace multiple occurrences in specific column

```
$ # replace space with underscore only in 3rd column
$ # ^(([^|]+\|){2} captures first two columns
$ # [^|]* zero or more non-column separator characters
$ # as long as match is found, command will be repeated on same input line
$ echo 'foo bar|a b c|1 2 3|xyz abc' | sed -E ':a s/^(([^|]+\|){2}[^|]*) /\1_/; ta'
foo bar|a b c|1_2_3|xyz abc

$ # using perl or awk might be simpler
$ # for ex: awk 'BEGIN{FS=OFS="|"} {gsub(/ /,"_",$3); print}'
```

- example to show difference between `b` and `t`

```
$ # whether or not 'R' is found on lines containing 'are', branch will happen
$ sed '/are/{s/R/*/g;b}; s/e/#/g' poem.txt
*oses are red,
Violets are blue,
Sugar is sw##t,
And so are you.

$ # branch only if line contains 'are' and substitution of 'R' succeeds
$ sed '/are/{s/R/*/g;t}; s/e/#/g' poem.txt
*oses are red,
Viol#ts ar# blu#,
Sugar is sw##t,
And so ar# you.
```

overlapping substitutions

- `t` command looping with label comes in handy for overlapping substitutions as well
- Note that in general this method will work recursively, see substitute recursively for example

```
$ # consider the problem of replacing empty columns with something
$ # case1: no consecutive empty columns - no problem
$ echo 'foo::bar::baz' | sed 's/::/:0:/g'
foo:0:bar:0:baz
$ # case2: consecutive empty columns are present - problematic
$ echo 'foo:::bar::baz' | sed 's/::/:0:/g'
foo:0::bar:0:baz

$ # t command looping will handle both cases
$ echo 'foo::bar::baz' | sed ':a s/::/:0:/; ta'
foo:0:bar:0:baz
$ echo 'foo:::bar::baz' | sed ':a s/::/:0:/; ta'
foo:0:0:bar:0:baz
```

Lines between two REGEXPs

- Simple cases were seen in address range section
- This section will deal with more cases and some corner cases

Include or Exclude matching REGEXPs

Consider the sample input file, for simplicity the two REGEXPs are **BEGIN** and **END** strings instead of regular expressions

```
$ cat range.txt
foo
BEGIN
1234
6789
END
bar
BEGIN
a
b
c
END
baz
```

First, lines between the two *REGEXP*s are to be printed

- Case 1: both starting and ending *REGEXP* part of output

```
$ sed -n '/BEGIN/,/END/p' range.txt
BEGIN
1234
6789
END
BEGIN
a
b
c
END
```

- Case 2: both starting and ending *REGEXP* not part of ouput

```
$ # remember that empty REGEXP section will reuse previously matched REGEXP
$ sed -n '/BEGIN/,/END/{//!p}' range.txt
1234
6789
a
b
c
```

- Case 3: only starting *REGEXP* part of output

```
$ sed -n '/BEGIN/,/END/{/END/!p}' range.txt
BEGIN
1234
6789
BEGIN
a
b
c
```

- Case 4: only ending *REGEXP* part of output

```
$ sed -n '/BEGIN/,/END/{/BEGIN/!p}' range.txt
1234
6789
END
a
b
c
END
```

Second, lines between the two *REGEXP*s are to be deleted

- Case 5: both starting and ending *REGEXP* not part of output

```
$ sed '/BEGIN/,/END/d' range.txt
foo
bar
baz
```

- Case 6: both starting and ending *REGEXP* part of output

```
$ # remember that empty REGEXP section will reuse previously matched REGEXP
$ sed '/BEGIN/,/END/{//!d}' range.txt
foo
BEGIN
END
bar
BEGIN
END
baz
```

- Case 7: only starting *REGEXP* part of output

```
$ sed '/BEGIN/,/END/{/BEGIN/!d}' range.txt
foo
BEGIN
bar
BEGIN
baz
```

- Case 8: only ending *REGEXP* part of output

```
$ sed '/BEGIN/,/END/{/END/!d}' range.txt
foo
END
bar
END
baz
```

First or Last block

- Getting first block is very simple by using `q` command

```
$ sed -n '/BEGIN/,/END/{p;/END/q}' range.txt
BEGIN
1234
6789
END

$ # use other tricks discussed in previous section as needed
$ sed -n '/BEGIN/,/END/{//!p;/END/q}' range.txt
1234
6789
```

- To get last block, reverse the input linewise, the order of *REGEXP*s and finally reverse again

```
$ tac range.txt | sed -n '/END/,/BEGIN/{p;/BEGIN/q}' | tac
BEGIN
a
b
c
END

$ # use other tricks discussed in previous section as needed
$ tac range.txt | sed -n '/END/,/BEGIN/{//!p;/BEGIN/q}' | tac
a
b
c
```

- To get a specific block, say 3rd one, `awk` or `perl` would be a better choice

Broken blocks

- If there are blocks with ending *REGEXP* but without corresponding starting *REGEXP*, `sed -n '/BEGIN/,/END/p'` will suffice
- Consider the modified input file where final starting *REGEXP* doesn't have corresponding ending

```
$ cat broken_range.txt
foo
BEGIN
1234
6789
END
bar
BEGIN
a
b
c
baz
```

- All lines till end of file gets printed with simple use of `sed -n '/BEGIN/,/END/p'`
- The file reversing trick comes in handy here as well
- But if both kinds of broken blocks are present, further processing will be required. Better to use `awk` or `perl` in such cases

```
$ sed -n '/BEGIN/,/END/p' broken_range.txt
BEGIN
1234
6789
END
BEGIN
a
b
c
baz

$ tac broken_range.txt | sed -n '/END/,/BEGIN/p' | tac
BEGIN
1234
6789
END
```

- If there are multiple starting *REGEXP* but single ending *REGEXP*, the reversing trick comes handy again

```
$ cat uneven_range.txt
foo
BEGIN
1234
BEGIN
42
6789
END
bar
BEGIN
a
BEGIN
b
BEGIN
c
BEGIN
d
BEGIN
e
END
baz

$ tac uneven_range.txt | sed -n '/END/,/BEGIN/p' | tac
BEGIN
42
6789
END
BEGIN
e
END
```

sed scripts

- `sed` commands can be placed in a file and called using `-f` option or directly executed using shebang)
- See sed manual - Some Sample Scripts for more examples
- See sed manual - Often-Used Commands for more details on using comments

```
$ cat script.sed
# each line is a command
/is/cfoo bar
/you/r 3.txt
/you/d
# single quotes can be used freely
s/are/'are'/g

$ sed -f script.sed poem.txt
Roses 'are' red,
Violets 'are' blue,
foo bar
3
13

$ # command line options are specified as usual
$ sed -nf script.sed poem.txt
foo bar
3
13
```

- command line options can be specified along with shebang as well as added at time of invocation
- **Note** usage of options along with shebang depends on lot of factors

```
$ type sed
sed is /bin/sed

$ cat executable.sed
#!/bin/sed -f
/is/cfoo bar
/you/r 3.txt
/you/d
s/are/'are'/g

$ chmod +x executable.sed

$ ./executable.sed poem.txt
Roses 'are' red,
Violets 'are' blue,
foo bar
3
13

$ ./executable.sed -n poem.txt
foo bar
3
13
```

Further Reading

- Manual and related
  - `man sed` and `info sed` for more details, known issues/limitations as well as options/commands not covered in this tutorial
  - GNU sed manual has even more detailed information and examples
  - sed FAQ, but last modified '10 March 2003'
  - BSD/macOS Sed vs GNU Sed vs the POSIX Sed specification
- Tutorials and Q&A
  - sed basics
  - sed detailed tutorial - has details on differences between various `sed` versions as well
  - sed one-liners explained
  - cheat sheet
  - common search and replace examples
  - sed Q&A on unix stackexchange
  - sed Q&A on stackoverflow
- Selected examples - portable solutions, commands not covered in this tutorial, same problem solved

  using different tools, etc
  - replace multiline string
  - deleting empty lines with optional white spaces
  - print only line above the matching line
  - How to select lines between two patterns?
  - get lines between two patterns only if there is third pattern between them
    - similar example
- Learn Regular Expressions
  - Regular Expressions Tutorial
  - regexcrossword
  - What does this regex mean?
- Related tools
  - sedsed - Debugger, indenter and HTMLizer for sed scripts
  - xo - composes regular expression match groups

GNU awk

Table of Contents

```
$ awk --version | head -n1
GNU Awk 4.1.3, API: 1.1 (GNU MPFR 3.1.4, GNU MP 6.1.0)

$ man awk
GAWK(1)                        Utility Commands                        GAWK(1)

NAME
       gawk - pattern scanning and processing language

SYNOPSIS
       gawk [ POSIX or GNU style options ] -f program-file [ -- ] file ...
       gawk [ POSIX or GNU style options ] [ -- ] program-text file ...

DESCRIPTION
       Gawk  is  the  GNU Project's implementation of the AWK programming lan-
       guage.  It conforms to the definition of  the  language  in  the  POSIX
       1003.1  Standard.   This version in turn is based on the description in
       The AWK Programming Language, by Aho, Kernighan, and Weinberger.   Gawk
       provides  the additional features found in the current version of Brian
       Kernighan's awk and a number of GNU-specific extensions.
...
```

Prerequisites and notes

- familiarity with programming concepts like variables, printing, control structures, arrays, etc
- familiarity with regular expressions
    - if not, check out **ERE** portion of GNU sed regular expressions which is close enough to features available in `gawk`
- this tutorial is primarily focussed on short programs that are easily usable from command line, similar to using `grep` , `sed` , etc
- see Gawk: Effective AWK Programming manual for complete reference, has information on other `awk` versions as well as notes on POSIX standard

Field processing

Default field separation

- `$0` contains the entire input record
  - default input record separator is newline character
- `$1` contains the first field text
  - default input field separator is one or more of continuous space, tab or newline characters
- `$2` contains the second field text and so on
- `$(2+3)` result of expressions can be used, this one evaluates to `$5` and hence gives fifth field
  - similarly if variable `i` has value `2`, then `$(i+3)` will give fifth field
  - See also gawk manual - Expressions
- `NF` is a built-in variable which contains number of fields in the current record
  - so, `$NF` will give last field
  - `$(NF-1)` will give second last field and so on

```
$ cat fruits.txt
fruit   qty
apple   42
banana  31
fig     90
guava   6

$ # print only first field
$ awk '{print $1}' fruits.txt
fruit
apple
banana
fig
guava

$ # print only second field
$ awk '{print $2}' fruits.txt
qty
42
31
90
6
```

Specifying different input field separator

- by using `-F` command line option
- by setting `FS` variable

- See FPAT and FIELDWIDTHS section for other ways of defining input fields

```
$ # second field where input field separator is :
$ echo 'foo:123:bar:789' | awk -F: '{print $2}'
123

$ # last field
$ echo 'foo:123:bar:789' | awk -F: '{print $NF}'
789

$ # first and last field
$ # note the use of , and space between output fields
$ echo 'foo:123:bar:789' | awk -F: '{print $1, $NF}'
foo 789

$ # second last field
$ echo 'foo:123:bar:789' | awk -F: '{print $(NF-1)}'
bar

$ # use quotes to avoid clashes with shell special characters
$ echo 'one;two;three;four' | awk -F';' '{print $3}'
three
```

- Regular expressions based input field separator

```
$ echo 'Sample123string54with908numbers' | awk -F'[0-9]+' '{print $2}'
string

$ # first field will be empty as there is nothing before '{'
$ echo '{foo}   bar=baz' | awk -F'[{}= ]+' '{print $1}'

$ echo '{foo}   bar=baz' | awk -F'[{}= ]+' '{print $2}'
foo
$ echo '{foo}   bar=baz' | awk -F'[{}= ]+' '{print $3}'
bar
```

- default input field separator is one or more of continuous space, tab or newline characters (will be termed as whitespace here on)
    - exact same behavior if `FS` is assigned single space character
- in addition, leading and trailing whitespaces won't be considered when splitting the input record

```
$ printf ' a    ate b\tc   \n'
 a    ate b     c
$ printf ' a    ate b\tc   \n' | awk '{print $1}'
a
$ printf ' a    ate b\tc   \n' | awk '{print NF}'
4
$ # same behavior if FS is assigned to single space character
$ printf ' a    ate b\tc   \n' | awk -F' ' '{print $1}'
a
$ printf ' a    ate b\tc   \n' | awk -F' ' '{print NF}'
4

$ # for anything else, leading/trailing whitespaces will be considered
$ printf ' a    ate b\tc   \n' | awk -F'[ \t]+' '{print $2}'
a
$ printf ' a    ate b\tc   \n' | awk -F'[ \t]+' '{print NF}'
6
```

- assigning empty string to FS will split the input record character wise
- note the use of command line option `-v` to set FS

```
$ echo 'apple' | awk -v FS= '{print $1}'
a
$ echo 'apple' | awk -v FS= '{print $2}'
p
$ echo 'apple' | awk -v FS= '{print $NF}'
e

$ # detecting multibyte characters depends on locale
$ printf 'hi how are you?'  | awk -v FS= '{print $3}'
```

Further Reading

- gawk manual - Field Splitting Summary
- stackoverflow - explanation on default FS
- unix.stackexchange - filter lines if it contains a particular character only once

Specifying different output field separator

- by setting `OFS` variable
- also gets added between every argument to `print` statement
  - use printf to avoid this

- default is single space

```
$ # statements inside BEGIN are executed before processing any input text
$ echo 'foo:123:bar:789' | awk 'BEGIN{FS=OFS=":"} {print $1, $NF}'
foo:789
$ # can also be set using command line option -v
$ echo 'foo:123:bar:789' | awk -F: -v OFS=':' '{print $1, $NF}'
foo:789

$ # changing a field will re-build contents of $0
$ echo ' a      ate b   ' | awk '{$2 = "foo"; print $0}' | cat -A
a foo b$

$ # $1=$1 is an idiomatic way to re-build when there is nothing else to change
$ echo 'foo:123:bar:789' | awk -F: -v OFS='-' '{print $0}'
foo:123:bar:789
$ echo 'foo:123:bar:789' | awk -F: -v OFS='-' '{$1=$1; print $0}'
foo-123-bar-789

$ # OFS is used to separate different arguments given to print
$ echo 'foo:123:bar:789' | awk -F: -v OFS='\t' '{print $1, $3}'
foo     bar

$ echo 'Sample123string54with908numbers' | awk -F'[0-9]+' '{$1=$1; print $0}'
Sample string with numbers
```

Filtering

Idiomatic print usage

- `print` statement with no arguments will print contents of `$0`
- if condition is specified without corresponding statements, contents of `$0` is printed if condition evaluates to true
- `1` is typically used to represent always true condition and thus print contents of `$0`

```
$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ # displaying contents of input file(s) similar to 'cat' command
$ # equivalent to using awk '{print $0}' and awk '1'
$ awk '{print}' poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.
```

Field comparison

- Each block of statements within `{}` can be prefixed by an optional condition so that those statements will execute only if condition evaluates to true
- Condition specified without corresponding statements will lead to printing contents of `$0` if condition evaluates to true

```
$ # if first field exactly matches the string 'apple'
$ awk '$1=="apple"{print $2}' fruits.txt
42

$ # print first field if second field > 35
$ # NR>1 to avoid the header line
$ # NR built-in variable contains record number
$ awk 'NR>1 && $2>35{print $1}' fruits.txt
apple
fig

$ # print header and lines with qty < 35
$ awk 'NR==1 || $2<35' fruits.txt
fruit   qty
banana  31
guava   6
```

- If the above examples are too confusing, think of it as syntactical sugar
- Statements are grouped within `{}`
  - inside `{}` , we have a `if` control structure
  - Like `c` language, braces not needed for single statements within `if` , but consider that `{}`

    is used for clarity
  - From this explicit syntax, remove the outer `{}` , `if` and `()` used for `if`
- As we'll see later, this allows to mash up few lines of program compactly on command line itself
  - Of course, for medium to large programs, it is better to put the code in separate file. See awk scripts section

```
$ # awk '$1=="apple"{print $2}' fruits.txt
$ awk '{
         if($1 == "apple"){
            print $2
         }
       }' fruits.txt
42

$ # awk 'NR==1 || $2<35' fruits.txt
$ awk '{
         if(NR==1 || $2<35){
            print $0
         }
       }' fruits.txt
fruit   qty
banana  31
guava   6
```

Further Reading

- gawk manual - Truth Values and Conditions
- gawk manual - Operator Precedence
- unix.stackexchange - filtering columns by header name

Regular expressions based filtering

- the *REGEXP* is specified within `//` and by default acts upon `$0`
- See also stackoverflow - lines around matching regexp

```
$ # all lines containing the string 'are'
$ # same as: grep 'are' poem.txt
$ awk '/are/' poem.txt
Roses are red,
Violets are blue,
And so are you.

$ # negating REGEXP, same as: grep -v 'are' poem.txt
$ awk '!/are/' poem.txt
Sugar is sweet,

$ # same as: grep 'are' poem.txt | grep -v 'so'
$ awk '/are/ && !/so/' poem.txt
Roses are red,
Violets are blue,

$ # lines starting with 'a' or 'b'
$ awk '/^[ab]/' fruits.txt
apple   42
banana  31

$ # print last field of all lines containing 'are'
$ awk '/are/{print $NF}' poem.txt
red,
blue,
you.
```

- strings can be used as well, which will be interpreted as *REGEXP* if necessary
- Allows using shell variables instead of hardcoded *REGEXP*
  - that section also notes difference between using `//` and string

```
$ awk '$0 !~ "are"' poem.txt
Sugar is sweet,

$ awk '$0 ~ "^[ab]"' fruits.txt
apple   42
banana  31

$ # also helpful if search strings have the / delimiter character
$ cat paths.txt
/foo/a/report.log
/foo/y/power.log
$ awk '/\/foo\/a\//' paths.txt
/foo/a/report.log
$ awk '$0 ~ "/foo/a/"' paths.txt
/foo/a/report.log
```

- *REGEXP* matching against specific field

```
$ # if first field contains 'a'
$ awk '$1 ~ /a/' fruits.txt
apple   42
banana  31
guava   6

$ # if first field contains 'a' and qty > 20
$ awk '$1 ~ /a/ && $2 > 20' fruits.txt
apple   42
banana  31

$ # if first field does NOT contain 'a'
$ awk '$1 !~ /a/' fruits.txt
fruit   qty
fig     90
```

Fixed string matching

- to search a string literally, `index` function can be used instead of *REGEXP*
  - similar to `grep -F`
- the function returns the starting position and `0` if no match found

```
$ cat eqns.txt
a=b,a+b=c,c*d
a+b,pi=3.14,5e12
i*(t+9-g)/8,4-a+b

$ # no output since '+' is meta character, would need '/a\+b/'
$ awk '/a+b/' eqns.txt
$ # same as: grep -F 'a+b' eqns.txt
$ awk 'index($0,"a+b")' eqns.txt
a+b,pi=3.14,5e12
i*(t+9-g)/8,4-a+b

$ # much easier than '/i\*\(t\+9-g\)/'
$ awk 'index($0,"i*(t+9-g)")' eqns.txt
i*(t+9-g)/8,4-a+b

$ # check only last field
$ awk -F, 'index($NF,"a+b")' eqns.txt
i*(t+9-g)/8,4-a+b
$ # index not needed if entire field/line is being compared
$ awk -F, '$1=="a+b"' eqns.txt
a+b,pi=3.14,5e12
```

- return value is useful to match at specific position
- for ex: at start/end of line

```
$ # start of line
$ awk 'index($0,"a+b")==1' eqns.txt
a+b,pi=3.14,5e12

$ # end of line
$ # length function returns number of characters, by default acts on $0
$ awk 'index($0,"a+b")==length()-length("a+b")+1' eqns.txt
i*(t+9-g)/8,4-a+b
$ # to avoid repetitions, save the search string in variable
$ awk -v s="a+b" 'index($0,s)==length()-length(s)+1' eqns.txt
i*(t+9-g)/8,4-a+b
```

Line number based filtering

- Built-in variable `NR` contains total records read so far
- Use `FNR` if you need line numbers separately for multiple file processing

```
$ # same as: head -n2 poem.txt | tail -n1
$ awk 'NR==2' poem.txt
Violets are blue,

$ # print 2nd and 4th line
$ awk 'NR==2 || NR==4' poem.txt
Violets are blue,
And so are you.

$ # same as: tail -n1 poem.txt
$ # statements inside END are executed after processing all input text
$ awk 'END{print}' poem.txt
And so are you.

$ awk 'NR==4{print $2}' fruits.txt
90
```

- for large input, use `exit` to avoid unnecessary record processing

```
$ seq 14323 14563435 | awk 'NR==234{print; exit}'
14556

$ # sample time comparison
$ time seq 14323 14563435 | awk 'NR==234{print; exit}'
14556

real    0m0.004s
user    0m0.004s
sys     0m0.000s
$ time seq 14323 14563435 | awk 'NR==234{print}'
14556

real    0m2.167s
user    0m2.280s
sys     0m0.092s
```

- See also unix.stackexchange - filtering list of lines from every X number of lines

Case Insensitive filtering

```
$ # same as: grep -i 'rose' poem.txt
$ awk -v IGNORECASE=1 '/rose/' poem.txt
Roses are red,

$ # for small enough set, can also use REGEXP character class
$ awk '/[rR]ose/' poem.txt
Roses are red,

$ # another way is to use built-in string function 'tolower'
$ awk 'tolower($0) ~ /rose/' poem.txt
Roses are red,
```

Changing record separators

- `RS` to change input record separator
- default is newline character

```
$ s='this is a sample string'

$ # space as input record separator, printing all records
$ printf "$s" | awk -v RS=' ' '{print NR, $0}'
1 this
2 is
3 a
4 sample
5 string

$ # print all records containing 'a'
$ printf "$s" | awk -v RS=' ' '/a/'
a
sample
```

- `ORS` to change output record separator
- gets added to every `print` statement
  - use printf to avoid this
- default is newline character

```
$ seq 3 | awk '{print $0}'
1
2
3
$ # note that there is empty line after last record
$ seq 3 | awk -v ORS='\n\n' '{print $0}'
1

2

3

$ # dynamically changing ORS
$ # can also use: seq 6 | awk '{ORS = NR%2 ? " " : RS} 1'
$ seq 6 | awk '{ORS = NR%2 ? " " : "\n"} 1'
1 2
3 4
5 6
$ seq 6 | awk '{ORS = NR%3 ? "-" : "\n"} 1'
1-2-3
4-5-6
```

Paragraph mode

- When `RS` is set to empty string, one or more consecutive empty lines is used as input record separator
- Can also use regular expression `RS=\n\n+` but there are subtle differences, see gawk manual - multiline records. Important points from that link quoted below

> However, there is an important difference between ‘RS = ""’ and ‘RS = "\n\n+"’. In the first case, leading newlines in the input data file are ignored, and if a file ends without extra blank lines after the last record, the final newline is removed from the record. In the second case, this special processing is not done
>
> Now that the input is separated into records, the second step is to separate the fields in the records. One way to do this is to divide each of the lines into fields in the normal manner. This happens by default as the result of a special feature. When RS is set to the empty string and FS is set to a single character, the newline character always acts as a field separator. This is in addition to whatever field separations result from FS
>
> When FS is the null string ("") or a regexp, this special feature of RS does not apply. It does apply to the default field separator of a single space: ‘FS = " "’

Consider the below sample file

```
$ cat sample.txt
Hello World

Good day
How are you

Just do-it
Believe it

Today is sunny
Not a bit funny
No doubt you like it too

Much ado about nothing
He he he
```

- Filtering paragraphs

```
$ # print all paragraphs containing 'it'
$ # if extra newline at end is undesirable, can use
$ # awk -v RS= '/it/{print c++ ? "\n" $0 : $0}' sample.txt
$ awk -v RS= -v ORS='\n\n' '/it/' sample.txt
Just do-it
Believe it

Today is sunny
Not a bit funny
No doubt you like it too

$ # based on number of lines in each paragraph
$ awk -F'\n' -v RS= -v ORS='\n\n' 'NF==1' sample.txt
Hello World

$ awk -F'\n' -v RS= -v ORS='\n\n' 'NF==2 && /do/' sample.txt
Just do-it
Believe it

Much ado about nothing
He he he
```

- Re-structuring paragraphs

```
$ # default FS is one or more of continuous space, tab or newline characters
$ # default OFS is single space
$ # so, $1=$1 will change it uniformly to single space between fields
$ awk -v RS= '{$1=$1} 1' sample.txt
Hello World
Good day How are you
Just do-it Believe it
Today is sunny Not a bit funny No doubt you like it too
Much ado about nothing He he he

$ # a better usecase
$ awk 'BEGIN{FS="\n"; OFS=". "; RS=""; ORS="\n\n"} {$1=$1} 1' sample.txt
Hello World

Good day. How are you

Just do-it. Believe it

Today is sunny. Not a bit funny. No doubt you like it too

Much ado about nothing. He he he
```

Further Reading

- unix.stackexchange - filtering line surrounded by empty lines
- stackoverflow - excellent example and explanation of RS and FS

Multicharacter RS

- Some marker like `Error` or `Warning` etc

```
$ cat report.log
blah blah
Error: something went wrong
more blah
whatever
Error: something surely went wrong
some text
some more text
blah blah blah

$ awk -v RS='Error:' 'END{print NR-1}' report.log
2
$ awk -v RS='Error:' 'NR==1' report.log
blah blah

$ # filter 'Error:' block matching particular string
$ # to preserve formatting, use: '/whatever/{print RS $0}'
$ awk -v RS='Error:' '/whatever/' report.log
 something went wrong
more blah
whatever

$ # blocks with more than 3 lines
$ # splitting string with 3 newlines will yield 4 fields
$ awk -F'\n' -v RS='Error:' 'NF>4{print RS $0}' report.log
Error: something surely went wrong
some text
some more text
blah blah blah
```

- Regular expression based `RS`
    - the `RT` variable will contain string matched by `RS`
- Note that entire input is treated as single string, so `^` and `$` anchors will apply only once - not every line

```
$ s='Sample123string54with908numbers'
$ printf "$s" | awk -v RS='[0-9]+' 'NR==1'
Sample

$ # note the relationship between record and separators
$ printf "$s" | awk -v RS='[0-9]+' '{print NR " : " $0 " - " RT}'
1 : Sample - 123
2 : string - 54
3 : with - 908
4 : numbers -

$ # need to be careful of empty records
$ printf '123string54with908' | awk -v RS='[0-9]+' '{print NR " : " $0}'
1 :
2 : string
3 : with
$ # and newline at end of input
$ printf '123string54with908\n' | awk -v RS='[0-9]+' '{print NR " : " $0}'
1 :
2 : string
3 : with
4 :
```

- Joining lines based on specific end of line condition

```
$ cat msg.txt
Hello there.
It will rain to-
day. Have a safe
and pleasant jou-
rney.

$ # join lines ending with - to next line
$ # by manipulating RS and ORS
$ awk -v RS='-\n' -v ORS= '1' msg.txt
Hello there.
It will rain today. Have a safe
and pleasant journey.

$ # by manipulating ORS alone, sub function covered in later sections
$ awk '{ORS = sub(/-$/,"") ? "" : "\n"} 1' msg.txt
Hello there.
It will rain today. Have a safe
and pleasant journey.
$ # easier: perl -pe 's/-\n//' msg.txt as newline is still part of input line
```

- processing null terminated input

```
$ printf 'foo\0bar\0' | cat -A
foo^@bar^@$
$ printf 'foo\0bar\0' | awk -v RS='\0' '{print}'
foo
bar
```

Further Reading

- gawk manual - Records
- unix.stackexchange - Slurp-mode in awk
- stackoverflow - using RS to count number of occurrences of a given string

Substitute functions

- Use `sub` string function for replacing first occurrence
- Use `gsub` for replacing all occurrences
- By default, `$0` which contains input record is modified, can specify any other field or variable as needed

```
$ # replacing first occurrence
$ echo '1-2-3-4-5' | awk '{sub("-", ":")} 1'
1:2-3-4-5

$ # replacing all occurrences
$ echo '1-2-3-4-5' | awk '{gsub("-", ":")} 1'
1:2:3:4:5

$ # return value for sub/gsub is number of replacements made
$ echo '1-2-3-4-5' | awk '{n=gsub("-", ":"); print n} 1'
4
1:2:3:4:5

$ # // format is better suited to specify search REGEXP
$ echo '1-2-3-4-5' | awk '{gsub(/[^-]+/, "abc")} 1'
abc-abc-abc-abc-abc

$ # replacing all occurrences only for third field
$ echo 'one;two;three;four' | awk -F';' '{gsub("e", "E", $3)} 1'
one two thrEE four
```

- Use `gensub` to return the modified string unlike `sub` or `gsub` which modifies inplace

- it also supports back-references and ability to modify specific match
- acts upon `$0` if target is not specified

```
$ # replace second occurrence
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(":", "-", 2)} 1'
foo:123-bar:baz
$ # use REGEXP as needed
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "XYZ", 2)} 1'
foo:XYZ:bar:baz

$ # or print the returned string directly
$ echo 'foo:123:bar:baz' | awk '{print gensub(":", "-", 2)}'
foo:123-bar:baz

$ # replace third occurrence
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "XYZ", 3)} 1'
foo:123:XYZ:baz

$ # replace all occurrences, similar to gsub
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "XYZ", "g")} 1'
XYZ:XYZ:XYZ:XYZ

$ # target other than $0
$ echo 'foo:123:bar:baz' | awk -F: -v OFS=: '{$1=gensub(/o/, "b", 2, $1)} 1'
fob:123:bar:baz
```

- back-reference examples
- use `\"` within double-quotes to represent `"` character in replacement string
- use `\\1` to represent `\1` - the first captured group and so on
- `&` or `\0` will back-reference entire matched string

```
$ # replacing last occurrence without knowing how many occurrences are there
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/(.*):/, "\\1-", 1)} 1'
foo:123:bar-baz
$ echo 'foo and bar and baz land good' | awk '{$0=gensub(/(.*)and/, "\\1XYZ", 1)} 1'
foo and bar and baz lXYZ good

$ # use word boundaries as necessary
$ echo 'foo and bar and baz land good' | awk '{$0=gensub(/(.*)\<and\>/, "\\1XYZ", 1)
} 1'
foo and bar XYZ baz land good

$ # replacing last but one
$ echo '456:foo:123:bar:789:baz' | awk '{$0=gensub(/(.*):(.*:)/, "\\1-\\2", 1)} 1'
456:foo:123:bar-789:baz

$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "\"&\"", "g")} 1'
"foo":"123":"bar":"baz"
```

- saving quotes in variables - to avoid escaping double quotes or having to use octal code for single quotes

```
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "\047&\047", "g")} 1'
'foo':'123':'bar':'baz'
$ echo 'foo:123:bar:baz' | awk -v sq="'" '{$0=gensub(/[^:]+/, sq"&"sq, "g")} 1'
'foo':'123':'bar':'baz'

$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "\"&\"", "g")} 1'
"foo":"123":"bar":"baz"
$ echo 'foo:123:bar:baz' | awk -v dq='"' '{$0=gensub(/[^:]+/, dq"&"dq, "g")} 1'
"foo":"123":"bar":"baz"
```

Further Reading

- gawk manual - String-Manipulation Functions
- gawk manual - escape processing

Inplace file editing

- Use this option with caution, preferably after testing that the `awk` code is working as intended

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

$ awk -i inplace '{gsub("e", "E")} 1' greeting.txt
$ cat greeting.txt
Hi thErE
HavE a nicE day
```

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

```
$ cat f1
I ate 3 apples
$ cat f2
I bought two bananas and 3 mangoes

$ awk -i inplace '{gsub("3", "three")} 1' f1 f2
$ cat f1
I ate three apples
$ cat f2
I bought two bananas and three mangoes
```

- to create backups of original file, set `INPLACE_SUFFIX` variable

```
$ awk -i inplace -v INPLACE_SUFFIX='.bkp' '{gsub("three", "3")} 1' f1
$ cat f1
I ate 3 apples
$ cat f1.bkp
I ate three apples
```

- See gawk manual - Enabling In-Place File Editing for implementation details

Using shell variables

- when `awk` code is part of shell program and shell variable needs to be passed as input to `awk` code
- for example:
    - command line argument passed to shell script, which is in turn passed on to `awk`
    - control structures in shell script calling `awk` with different search strings
- See also stackoverflow - How do I use shell variables in an awk script?

```
$ # examples tested with bash shell

$ f='apple'
$ awk -v word="$f" '$1==word' fruits.txt
apple   42
$ f='fig'
$ awk -v word="$f" '$1==word' fruits.txt
fig     90

$ q='20'
$ awk -v threshold="$q" 'NR==1 || $2>threshold' fruits.txt
fruit   qty
apple   42
banana  31
fig     90
```

- accessing shell environment variables

```
$ # existing environment variable
$ awk 'BEGIN{print ENVIRON["PWD"]}'
/home/learnbyexample
$ awk 'BEGIN{print ENVIRON["SHELL"]}'
/bin/bash

$ # defined along with awk code
$ word='hello world' awk 'BEGIN{print ENVIRON["word"]}'
hello world

$ # using ENVIRON also prevents awk's interpretation of escape sequences
$ s='a\n=c'
$ foo="$s" awk 'BEGIN{print ENVIRON["foo"]}'
a\n=c
$ awk -v foo="$s" 'BEGIN{print foo}'
a
=c
```

- passing *REGEXP*
- See also gawk manual - Using Dynamic Regexps

```
$ s='are'
$ # for: awk '!/are/' poem.txt
$ awk -v s="$s" '$0 !~ s' poem.txt
Sugar is sweet,
$ # for: awk '/are/ && !/so/' poem.txt
$ awk -v s="$s" '$0 ~ s && !/so/' poem.txt
Roses are red,
Violets are blue,

$ r='[^-]+'
$ echo '1-2-3-4-5' | awk -v r="$r" '{gsub(r, "abc")} 1'
abc-abc-abc-abc-abc

$ # escape sequence has to be doubled when string is interpreted as REGEXP
$ s='foo and bar and baz land good'
$ echo "$s" | awk '{$0=gensub("(.*)\\<and\\>", "\\1XYZ", 1)} 1'
foo and bar XYZ baz land good
$ # hence passing as variable should be
$ r='(.*)\\<and\\>'
$ echo "$s" | awk -v r="$r" '{$0=gensub(r, "\\1XYZ", 1)} 1'
foo and bar XYZ baz land good

$ # or use ENVIRON
$ r='(.*)\<and\>'
$ echo "$s" | r="$r" awk '{$0=gensub(ENVIRON["r"], "\\1XYZ", 1)} 1'
foo and bar XYZ baz land good
```

Multiple file input

- Example to show difference between `NR` and `FNR`

```
$ # NR for overall record number
$ awk 'NR==1' poem.txt greeting.txt
Roses are red,

$ # FNR for individual file's record number
$ # same as: head -q -n1 poem.txt greeting.txt
$ awk 'FNR==1' poem.txt greeting.txt
Roses are red,
Hi thErE
```

- Constructs to do some processing before starting each file as well as at the end
- `BEGINFILE` - to add code to be executed before start of each input file

- `ENDFILE` - to add code to be executed after processing each input file
- `FILENAME` - file name of current input file being processed

```
$ # similar to: tail -n1 poem.txt greeting.txt
$ awk 'BEGINFILE{print "file: "FILENAME}
       ENDFILE{print $0"\n------"}' poem.txt greeting.txt
file: poem.txt
And so are you.
------
file: greeting.txt
HavE a nicE day
------
```

- And of course, there can be usual `awk` code

```
$ awk 'BEGINFILE{print "file: "FILENAME}
       FNR==1;
       ENDFILE{print "------"}' poem.txt greeting.txt
file: poem.txt
Roses are red,
------
file: greeting.txt
Hi thErE
------

$ awk 'BEGINFILE{c++; print "file: "FILENAME}
       FNR==2;
       END{print "\nTotal input files: "c}' poem.txt greeting.txt
file: poem.txt
Violets are blue,
file: greeting.txt
HavE a nicE day

Total input files: 2
```

Further Reading

- gawk manual - Using ARGC and ARGV
- gawk manual - ARGIND
- gawk manual - ERRNO
- stackoverflow - Finding common value across multiple files

Control Structures

- Syntax is similar to `C` language and single statements inside control structures don't require to be grouped within `{}`
- See gawk manual - Control Statements for details

Remember that by default there is a loop that goes over all input records and constructs like `BEGIN` and `END` fall outside that loop

```
$ cat nums.txt
42
-2
10101
-3.14
-75
$ awk '{sum += $1} END{print sum}' nums.txt
10062.9

$ # uninitialized variables will have empty string
$ printf '' | awk '{sum += $1} END{print sum}'

$ # so either add '0' or use unary '+' operator to convert to number
$ printf '' | awk '{sum += $1} END{print +sum}'
0
```

if-else and loops

- We have already seen simple `if` examples in Filtering section
- See also gawk manual - Switch

```
$ # same as: sed -n '/are/ s/so/SO/p' poem.txt
$ # remember that sub/gsub returns number of substitutions made
$ awk '/are/{if(sub("so", "SO")) print}' poem.txt
And SO are you.
$ # of course, can also use
$ awk '/are/ && sub("so", "SO")' poem.txt
And SO are you.

$ # if-else example
$ awk 'NR>1{if($2>40) $0="+"$0; else $0="-"$0} 1' fruits.txt
fruit   qty
+apple   42
-banana  31
+fig     90
-guava   6
```

- conditional operator
- See also stackoverflow - finding min and max value of a column

```
$ cat nums.txt
42
-2
10101
-3.14
-75

$ # changing -ve to +ve and vice versa
$ # same as: awk '{if($0 ~ /^-/) sub(/^-/,""); else sub(/^/,"-")} 1' nums.txt
$ awk '{$0 ~ /^-/ ? sub(/^-/,"") : sub(/^/,"-")} 1' nums.txt
-42
2
-10101
3.14
75
$ # can also use: awk '!sub(/^-/,""){sub(/^/,"-")} 1' nums.txt
```

- for loop
- similar to `C` language, `break` and `continue` statements are also available
- See also stackoverflow - find missing numbers from sequential list

```
$ awk 'BEGIN{for(i=2; i<11; i+=2) print i}'
2
4
6
8
10

$ # looping each field
$ s='scat:cat:no cat:abdicate:cater'
$ echo "$s" | awk -F: -v OFS=: '{for(i=1;i<=NF;i++) if($i=="cat") $i="CAT"} 1'
scat:CAT:no cat:abdicate:cater
$ # can also use sub function
$ echo "$s" | awk -F: -v OFS=: '{for(i=1;i<=NF;i++) sub(/^cat$/,"CAT",$i)} 1'
scat:CAT:no cat:abdicate:cater
```

- while loop
- do-while is also available

```
$ awk 'BEGIN{i=2; while(i<11){print i; i+=2}}'
2
4
6
8
10

$ # recursive substitution
$ # here again return value of sub/gsub is useful
$ echo 'titillate' | awk '{while( gsub(/til/, "") ) print}'
tilate
ate
```

next and nextfile

- `next` will skip rest of statements and start processing next line of current file being processed
    - there is a loop by default which goes over all input records, `next` is applicable for that
    - it is similar to `continue` statement within loops
- it is often used in Two file processing

```
$ # here 'next' is used to skip processing header line
$ awk 'NR==1{print; next} /a.*a/{$0="*"$0} /[eiou]/{$0="-"$0} 1' fruits.txt
fruit   qty
-apple   42
*banana  31
-fig     90
-*guava   6
```

- `nextfile` is useful to skip remaining lines from current file being processed and move on to next file

```
$ # same as: head -q -n1 poem.txt greeting.txt fruits.txt
$ awk 'FNR>1{nextfile} 1' poem.txt greeting.txt fruits.txt
Roses are red,
Hi thErE
fruit   qty

$ # specific field
$ awk 'FNR>2{nextfile} {print $1}' poem.txt greeting.txt fruits.txt
Roses
Violets
Hi
HavE
fruit
apple

$ # similar to 'grep -il'
$ awk -v IGNORECASE=1 '/red/{print FILENAME; nextfile}' *
colors_1.txt
colors_2.txt
poem.txt
$ awk -v IGNORECASE=1 '$1 ~ /red/{print FILENAME; nextfile}' *
colors_1.txt
colors_2.txt
```

Multiline processing

- Processing consecutive lines

```
$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ # match two consecutive lines
$ awk 'p~/are/ && /is/{print p ORS $0} {p=$0}' poem.txt
Violets are blue,
Sugar is sweet,
$ # if only the second line is needed
$ awk 'p~/are/ && /is/; {p=$0}' poem.txt
Sugar is sweet,

$ # match three consecutive lines
$ awk 'p2~/red/ && p1~/blue/ && /is/{print p2} {p2=p1; p1=$0}' poem.txt
Roses are red,

$ # common mistake
$ sed -n '/are/{N;/is/p}' poem.txt
$ # would need something like this and not practical to extend for other cases
$ sed '$!N; /are.*\n.*is/p; D' poem.txt
Violets are blue,
Sugar is sweet,
```

Consider this sample input file

```
$ cat range.txt
foo
BEGIN
1234
6789
END
bar
BEGIN
a
b
c
END
baz
```

- extracting lines around matching line
- See also stackoverflow - lines around matching regexp
- how `n && n--` works:
    - need to note that right hand side of `&&` is processed only if left hand side is `true`

- so for example, if initially `n=2`, then we get
    - `2 && 2; n=1` - evaluates to `true`
    - `1 && 1; n=0` - evaluates to `true`
    - `0 &&` - evaluates to `false` ... no decrementing `n` and hence will be `false` until `n` is re-assigned non-zero value

```
$ # similar to: grep --no-group-separator -A1 'BEGIN' range.txt
$ awk '/BEGIN/{n=2} n && n--' range.txt
BEGIN
1234
BEGIN
a

$ # only print the line after matching line
$ # can also use: awk '/BEGIN/{n=1; next} n && n--' range.txt
$ awk 'n && n--; /BEGIN/{n=1}' range.txt
1234
a
$ # generic case: print nth line after match
$ awk 'n && !--n; /BEGIN/{n=3}' range.txt
END
c

$ # print second line prior to matched line
$ awk '/END/{print p2} {p2=p1; p1=$0}' range.txt
1234
b
$ # save all lines in an array for generic case
$ awk '/END/{print a[NR-3]} {a[NR]=$0}' range.txt
BEGIN
a
$ # or use the reversing trick
$ tac range.txt | awk 'n && !--n; /END/{n=3}' | tac
BEGIN
a
```

- Checking if multiple strings are present at least once in entire input file
- If there are lots of strings to check, use arrays

```
$ # can also use BEGINFILE instead of FNR==1
$ awk 'FNR==1{s1=s2=0} /is/{s1=1} /are/{s2=1} s1&&s2{print FILENAME; nextfile}' *
poem.txt
sample.txt

$ awk 'FNR==1{s1=s2=0} /foo/{s1=1} /report/{s2=1} s1&&s2{print FILENAME; nextfile}'
*
paths.txt
```

Two file processing

- We'll use awk's associative arrays (key-value pairs) here
  - key can be number or string
  - See also gawk manual - Arrays
- Unlike comm the input files need not be sorted and comparison can be done based on certain field(s) as well

Comparing whole lines

Consider the following test files

```
$ cat colors_1.txt
Blue
Brown
Purple
Red
Teal
Yellow

$ cat colors_2.txt
Black
Blue
Green
Red
White
```

- common lines and lines unique to one of the files
- For two files as input, `NR==FNR` will be true only when first file is being processed
- Using `next` will skip rest of code when first file is processed
- `a[$0]` will create unique keys (here entire line content is used as key) in array `a`

    - just referencing a key will create it if it doesn't already exist, with value as empty string (will also act as zero in numeric context)
- `$0 in a` will be true if key already exists in array `a`

```
$ # common lines
$ # same as: grep -Fxf colors_1.txt colors_2.txt
$ awk 'NR==FNR{a[$0]; next} $0 in a' colors_1.txt colors_2.txt
Blue
Red

$ # lines from colors_2.txt not present in colors_1.txt
$ # same as: grep -vFxf colors_1.txt colors_2.txt
$ awk 'NR==FNR{a[$0]; next} !($0 in a)' colors_1.txt colors_2.txt
Black
Green
White

$ # reversing the order of input files gives
$ # lines from colors_1.txt not present in colors_2.txt
$ awk 'NR==FNR{a[$0]; next} !($0 in a)' colors_2.txt colors_1.txt
Brown
Purple
Teal
Yellow
```

Comparing specific fields

Consider the sample input file

```
$ cat marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59
ECE     Om      92
CSE     Amy     67
```

- single field
- For ex: only first field comparison by using `$1` instead of `$0` as key

```
$ cat list1
ECE
CSE

$ # extract only lines matching first field specified in list1
$ awk 'NR==FNR{a[$1]; next} $1 in a' list1 marks.txt
ECE     Raj     53
ECE     Joel    72
CSE     Surya   81
ECE     Om      92
CSE     Amy     67

$ # if header is needed as well
$ awk 'NR==FNR{a[$1]; next} FNR==1 || $1 in a' list1 marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
CSE     Surya   81
ECE     Om      92
CSE     Amy     67
```

- multiple fields
- create a string by adding some character between the fields to act as key
    - for ex: to avoid matching two field values `abc` and `123` to match with two other field values `ab` and `c123`
    - by adding character, say `_` , the key would be `abc_123` for first case and `ab_c123` for second case
    - this can still lead to false match if input data has `_`
    - there is also a built-in way to do this using gawk manual - Multidimensional Arrays

```
$ cat list2
EEE Moi
CSE Amy
ECE Raj

$ # extract only lines matching both fields specified in list2
$ awk 'NR==FNR{a[$1"_"$2]; next} $1"_"$2 in a' list2 marks.txt
ECE     Raj     53
EEE     Moi     68
CSE     Amy     67

$ # uses SUBSEP as separator, whose default value is non-printing character \034
$ awk 'NR==FNR{a[$1,$2]; next} ($1,$2) in a' list2 marks.txt
ECE     Raj     53
EEE     Moi     68
CSE     Amy     67
```

- field and value comparison

```
$ cat list3
ECE 70
EEE 65
CSE 80

$ # extract line matching Dept and minimum marks specified in list3
$ awk 'NR==FNR{d[$1]; m[$1]=$2; next} $1 in d && $3 >= m[$1]' list3 marks.txt
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
ECE     Om      92
```

getline

- If entire line (instead of fields) from one file is needed to change the other file, using `getline` would be faster
- But use it with caution
  - gawk manual - getline for details, especially about corner cases, errors, etc
  - gawk manual - Closing Input and Output Redirections if you have to start from beginning of file again

```
$ # replace mth line in poem.txt with nth line from nums.txt
$ awk -v m=3 -v n=2 'BEGIN{while(n-- > 0) getline s < "nums.txt"}
                     FNR==m{$0=s} 1' poem.txt
Roses are red,
Violets are blue,
-2
And so are you.

$ # without getline, but slower due to NR==FNR check for every line processed
$ awk -v m=3 -v n=2 'NR==FNR{if(FNR==n){s=$0; nextfile} next}
                     FNR==m{$0=s} 1' nums.txt poem.txt
Roses are red,
Violets are blue,
-2
And so are you.
```

- Another use case is if two files are to be processed exactly for same line numbers

```
$ # print line from fruits.txt if corresponding line from nums.txt is +ve number
$ awk -v file='nums.txt' '{getline num < file; if(num>0) print}' fruits.txt
fruit   qty
banana  31

$ # without getline, but has to save entire file in array
$ awk 'NR==FNR{n[FNR]=$0; next} n[FNR]>0' nums.txt fruits.txt
fruit   qty
banana  31
```

Further Reading

- stackoverflow - Fastest way to find lines of a text file from another larger text file
- unix.stackexchange - filter lines based on line numbers specified in another file
- stackoverflow - three file processing to extract a matrix subset
- unix.stackexchange - column wise merging
- stackoverflow - extract specific rows from a text file using an index file

Creating new fields

- Number of fields in input record can be changed by simply manipulating `NF`

```
$ # reducing fields
$ echo 'foo,bar,123,baz' | awk -F, -v OFS=, '{NF=2} 1'
foo,bar

$ # creating new empty field(s)
$ echo 'foo,bar,123,baz' | awk -F, -v OFS=, '{NF=5} 1'
foo,bar,123,baz,

$ # assigning to field greater than NF will create empty fields as needed
$ echo 'foo,bar,123,baz' | awk -F, -v OFS=, '{$7=42} 1'
foo,bar,123,baz,,,42

$ # adding a new 'Grade' field
$ awk 'BEGIN{OFS="\t"; g[9]="S"; g[8]="A"; g[7]="B"; g[6]="C"; g[5]="D"}
      {NF++; if(NR==1)$NF="Grade"; else $NF=g[int($(NF-1)/10)]} 1' marks.txt
Dept    Name    Marks   Grade
ECE     Raj     53      D
ECE     Joel    72      B
EEE     Moi     68      C
CSE     Surya   81      A
EEE     Tia     59      D
ECE     Om      92      S
CSE     Amy     67      C
```

- two file example

```
$ cat list4
Raj class_rep
Amy sports_rep
Tia placement_rep

$ awk -v OFS='\t' 'NR==FNR{r[$1]=$2; next}
         {NF++; if(FNR==1)$NF="Role"; else $NF=r[$2]} 1' list4 marks.txt
Dept    Name    Marks   Role
ECE     Raj     53      class_rep
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59      placement_rep
ECE     Om      92
CSE     Amy     67      sports_rep
```

Dealing with duplicates

- default value of uninitialized variable is `0` in numeric context and empty string in text context
  - and evaluates to `false` when used conditionally

Illustration to show default numeric value and array in action

```
$ printf 'mad\n42\n42\ndam\n42\n'
mad
42
42
dam
42

$ printf 'mad\n42\n42\ndam\n42\n' | awk '{print $0 "\t" int(a[$0]); a[$0]++}'
mad     0
42      0
42      1
dam     0
42      2
$ # only those entries with second column value zero will be retained
$ printf 'mad\n42\n42\ndam\n42\n' | awk '!a[$0]++'
mad
42
dam
```

- first, examples that retain only first copy of duplicates

```
$ cat duplicates.txt
abc  7   4
food toy ****
abc  7   4
test toy 123
good toy ****

$ # whole line
$ awk '!seen[$0]++' duplicates.txt
abc  7   4
food toy ****
test toy 123
good toy ****

$ # particular column
$ awk '!seen[$2]++' duplicates.txt
abc  7   4
food toy ****
```

- For multiple fields, separate them using `,` or form a string with some character in between

  - choose a character unlikely to appear in input data, else there can be false matches

```
$ awk '!seen[$2"_"$3]++' duplicates.txt
abc  7   4
food toy ****
test toy 123

$ # can also use simulated multidimensional array
$ # SUBSEP, whose default is \034 non-printing character, is used as separator
$ awk '!seen[$2,$3]++' duplicates.txt
abc  7   4
food toy ****
test toy 123
```

- retaining specific numbered copy

```
$ # second occurrence of duplicate
$ awk '++seen[$2]==2' duplicates.txt
abc  7   4
test toy 123

$ # third occurrence of duplicate
$ awk '++seen[$2]==3' duplicates.txt
good toy ****
```

- retaining only last copy of duplicate

```
$ # reverse the input line-wise, retain first copy and then reverse again
$ tac duplicates.txt | awk '!seen[$2]++' | tac
abc  7   4
good toy ****
```

- filtering based on duplicate count
- allows to emulate uniq command for specific fields
- See also unix.stackexchange - retain only parent directory paths

```
$ # all duplicates based on 1st column
$ awk 'NR==FNR{a[$1]++; next} a[$1]>1' duplicates.txt duplicates.txt
abc  7   4
abc  7   4
$ # all duplicates based on 3rd column
$ awk 'NR==FNR{a[$3]++; next} a[$3]>1' duplicates.txt duplicates.txt
abc  7   4
food toy ****
abc  7   4
good toy ****

$ # more than 2 duplicates based on 2nd column
$ awk 'NR==FNR{a[$2]++; next} a[$2]>2' duplicates.txt duplicates.txt
food toy ****
test toy 123
good toy ****

$ # only unique lines based on 3rd column
$ awk 'NR==FNR{a[$3]++; next} a[$3]==1' duplicates.txt duplicates.txt
test toy 123
```

Lines between two REGEXPs

- This section deals with filtering lines bound by two *REGEXP*s (referred to as blocks)
- For simplicity the two *REGEXP*s usually used in below examples are the strings **BEGIN** and **END**

All unbroken blocks

Consider the below sample input file, which doesn't have any unbroken blocks (i.e **BEGIN** and **END** are always present in pairs)

```
$ cat range.txt
foo
BEGIN
1234
6789
END
bar
BEGIN
a
b
c
END
baz
```

- Extracting lines between starting and ending *REGEXP*

```
$ # include both starting/ending REGEXP
$ # can also use: awk '/BEGIN/,/END/' range.txt
$ # which is similar to sed -n '/BEGIN/,/END/p'
$ # but not suitable to extend for other cases
$ awk '/BEGIN/{f=1} f; /END/{f=0}' range.txt
BEGIN
1234
6789
END
BEGIN
a
b
c
END

$ # exclude both starting/ending REGEXP
$ # can also use: awk '/BEGIN/{f=1; next} /END/{f=0} f' range.txt
$ awk '/END/{f=0} f; /BEGIN/{f=1}' range.txt
1234
6789
a
b
c
```

- Include only start or end *REGEXP*

```
$ # include only starting REGEXP
$ awk '/BEGIN/{f=1} /END/{f=0} f' range.txt
BEGIN
1234
6789
BEGIN
a
b
c

$ # include only ending REGEXP
$ awk 'f; /END/{f=0} /BEGIN/{f=1}' range.txt
1234
6789
END
a
b
c
END
```

- Extracting lines other than lines between the two *REGEXP*s

```
$ awk '/BEGIN/{f=1} !f; /END/{f=0}' range.txt
foo
bar
baz

$ # the other three cases would be
$ awk '/END/{f=0} !f; /BEGIN/{f=1}' range.txt
$ awk '!f; /BEGIN/{f=1} /END/{f=0}' range.txt
$ awk '/BEGIN/{f=1} /END/{f=0} !f' range.txt
```

Specific blocks

- Getting first block

```
$ awk '/BEGIN/{f=1} f; /END/{exit}' range.txt
BEGIN
1234
6789
END

$ # use other tricks discussed in previous section as needed
$ awk '/END/{exit} f; /BEGIN/{f=1}' range.txt
1234
6789
```

- Getting last block

```
$ # reverse input linewise, change the order of REGEXPs, finally reverse again
$ tac range.txt | awk '/END/{f=1} f; /BEGIN/{exit}' | tac
BEGIN
a
b
c
END

$ # or, save the blocks in a buffer and print the last one alone
$ # ORS contains output record separator, which is newline by default
$ seq 30 | awk '/4/{f=1; b=$0; next} f{b=b ORS $0} /6/{f=0} END{print b}'
24
25
26
```

- Getting blocks based on a counter

```
$ # all blocks
$ seq 30 | sed -n '/4/,/6/p'
4
5
6
14
15
16
24
25
26

$ # get only 2nd block
$ # can also use: seq 30 | awk -v b=2 '/4/{c++} c==b{print; if(/6/) exit}'
$ seq 30 | awk -v b=2 '/4/{c++} c==b; /6/ && c==b{exit}'
14
15
16

$ # to get all blocks greater than 'b' blocks
$ seq 30 | awk -v b=1 '/4/{f=1; c++} f && c>b; /6/{f=0}'
14
15
16
24
25
26
```

- excluding a particular block

```
$ # excludes 2nd block
$ seq 30 | awk -v b=2 '/4/{f=1; c++} f && c!=b; /6/{f=0}'
4
5
6
24
25
26
```

Broken blocks

- If there are blocks with ending *REGEXP* but without corresponding start, `awk '/BEGIN/{f=1} f; /END/{f=0}'` will suffice

- Consider the modified input file where starting *REGEXP* doesn't have corresponding ending

```
$ cat broken_range.txt
foo
BEGIN
1234
6789
END
bar
BEGIN
a
b
c
baz

$ # the file reversing trick comes in handy here as well
$ tac broken_range.txt | awk '/END/{f=1} f; /BEGIN/{f=0}' | tac
BEGIN
1234
6789
END
```

- But if both kinds of broken blocks are present, accumulate the records and print accordingly

```
$ cat multiple_broken.txt
qqqqqqq
BEGIN
foo
BEGIN
1234
6789
END
bar
END
0-42-1
BEGIN
a
BEGIN
b
END
;as;s;sd;

$ awk '/BEGIN/{f=1; buf=$0; next}
       f{buf=buf ORS $0}
       /END/{f=0; if(buf) print buf; buf=""}' multiple_broken.txt
BEGIN
1234
6789
END
BEGIN
b
END
```

Further Reading

- stackoverflow - select lines between two regexps
- unix.stackexchange - print only blocks with lines > n
- unix.stackexchange - print a block only if it contains matching string
- unix.stackexchange - print a block matching two different strings

Arrays

We've already seen examples using arrays, some more examples discussed in this section

- array looping

```
$ # average marks for each department
$ awk 'NR>1{d[$1]+=$3; c[$1]++} END{for(i in d)print i, d[i]/c[i]}' marks.txt
ECE 72.3333
EEE 63.5
CSE 74
```

- Sorting
- See gawk manual - Predefined Array Scanning Orders for more details

```
$ # by default, keys are traversed in random order
$ awk 'BEGIN{a["z"]=1; a["x"]=12; a["b"]=42; for(i in a)print i, a[i]}'
x 12
z 1
b 42

$ # index sorted ascending order as strings
$ awk 'BEGIN{PROCINFO["sorted_in"] = "@ind_str_asc";
       a["z"]=1; a["x"]=12; a["b"]=42; for(i in a)print i, a[i]}'
b 42
x 12
z 1

$ # value sorted ascending order as numbers
$ awk 'BEGIN{PROCINFO["sorted_in"] = "@val_num_asc";
       a["z"]=1; a["x"]=12; a["b"]=42; for(i in a)print i, a[i]}'
z 1
x 12
b 42
```

- deleting array elements

```
$ cat list5
CSE     Surya   75
EEE     Jai     69
ECE     Kal     83

$ # update entry if a match is found
$ # else append the new entries
$ awk '{ky=$1"_"$2} NR==FNR{upd[ky]=$0; next}
        ky in upd{$0=upd[ky]; delete upd[ky]} 1;
        END{for(i in upd)print upd[i]}' list5 marks.txt
Dept    Name    Marks
ECE     Raj     53
ECE     Joel    72
EEE     Moi     68
CSE     Surya   75
EEE     Tia     59
ECE     Om      92
CSE     Amy     67
ECE     Kal     83
EEE     Jai     69
```

- true multidimensional arrays
- length of sub-arrays need not be same. See gawk manual - Arrays of Arrays for details

```
$ awk 'NR>1{d[$1][$2]=$3} END{for(i in d["ECE"])print i}' marks.txt
Joel
Raj
Om

$ awk -v f='CSE' 'NR>1{d[$1][$2]=$3} END{for(i in d[f])print i, d[f][i]}' marks.txt
Surya 81
Amy 67
```

Further Reading

- gawk manual - all array topics
- unix.stackexchange - count words based on length
- unix.stackexchange - filtering specific lines

awk scripts

- For larger programs, save the code in a file and use `-f` command line option
- `;` is not needed to terminate a statement

- See also gawk manual - Command-Line Options for other related options

```
$ cat buf.awk
/BEGIN/{
    f=1
    buf=$0
    next
}

f{
    buf=buf ORS $0
}

/END/{
    f=0
    if(buf)
        print buf
    buf=""
}

$ awk -f buf.awk multiple_broken.txt
BEGIN
1234
6789
END
BEGIN
b
END
```

- Another advantage is that single quotes can be freely used

```
$ echo 'foo:123:bar:baz' | awk '{$0=gensub(/[^:]+/, "\047&\047", "g")} 1'
'foo':'123':'bar':'baz'

$ cat quotes.awk
{
    $0 = gensub(/[^:]+/, "'&'", "g")
}

1

$ echo 'foo:123:bar:baz' | awk -f quotes.awk
'foo':'123':'bar':'baz'
```

- If the code has been first tried out on command line, add `-o` option to get a pretty printed version

```
$ awk -o -v OFS='\t' 'NR==FNR{r[$1]=$2; next}
         {NF++; if(FNR==1)$NF="Role"; else $NF=r[$2]} 1' list4 marks.txt
Dept    Name    Marks   Role
ECE     Raj     53      class_rep
ECE     Joel    72
EEE     Moi     68
CSE     Surya   81
EEE     Tia     59      placement_rep
ECE     Om      92
CSE     Amy     67      sports_rep
```

File name can be passed along `-o` option, otherwise by default `awkprof.out` will be used

```
$ cat awkprof.out
        # gawk profile, created Tue Oct 24 15:10:02 2017

        # Rule(s)

        NR == FNR {
                r[$1] = $2
                next
        }

        {
                NF++
                if (FNR == 1) {
                        $NF = "Role"
                } else {
                        $NF = r[$2]
                }
        }

        1 {
                print $0
        }

$ # note that other command line options have to be provided as usual
$ # for ex: awk -v OFS='\t' -f awkprof.out list4 marks.txt
```

Miscellaneous

FPAT and FIELDWIDTHS

- `FS` allows to define field separator
- In contrast, `FPAT` allows to define what should the fields be made up of
- See also gawk manual - Defining Fields by Content

```
$ s='Sample123string54with908numbers'
$ # define fields to be one or more consecutive digits
$ echo "$s" | awk -v FPAT='[0-9]+' '{print $1, $2, $3}'
123 54 908
$ # define fields to be one or more consecutive alphabets
$ echo "$s" | awk -v FPAT='[a-zA-Z]+' '{print $1, $2, $3, $4}'
Sample string with numbers
```

- For simpler **csv** input having quoted strings if fields themselves have `,` in them, using `FPAT` is reasonable approach
- Use a proper parser if input can have other cases like newlines in fields
    - See unix.stackexchange - using csv parser for a sample program in `perl`

```
$ s='foo,"bar,123",baz,abc'
$ echo "$s" | awk -F, '{print $2}'
"bar
$ echo "$s" | awk -v FPAT='"[^"]*"|[^,]*' '{print $2}'
"bar,123"
```

- if input has well defined fields based on number of characters, `FIELDWIDTHS` can be used to specify width of each field

```
$ awk -v FIELDWIDTHS='8 3' -v OFS= '/fig/{$2=35} 1' fruits.txt
fruit   qty
apple   42
banana  31
fig     35
guava   6

$ # without FIELDWIDTHS
$ awk '/fig/{$2=35} 1' fruits.txt
fruit   qty
apple   42
banana  31
fig 35
guava   6
```

Further Reading

- unix.stackexchange - Modify records in fixed-width files
- unix.stackexchange - detecting empty fields in fixed width files
- stackoverflow - count number of times value is repeated each line

String functions

- `length` function - returns length of string, by default acts on `$0`

```
$ seq 8 13 | awk 'length()==1'
8
9

$ awk 'NR==1 || length($1)>4' fruits.txt
fruit   qty
apple   42
banana  31
guava   6

$ # character count and not byte count is calculated, similar to 'wc -m'
$ printf 'hi'  | awk '{print length()}'
3

$ # use -b option if number of bytes are needed
$ printf 'hi'  | awk -b '{print length()}'
6
```

- `split` function - similar to `FS` splitting input record into fields
- use `patsplit` function to get results similar to `FPAT`
- See also gawk manual - Split function
- See also unix.stackexchange - delimit second column

```
$ # 1st argument is string to be split
$ # 2nd argument is array to save results, indexed from 1
$ # 3rd argument is separator, default is FS
$ s='foo,1996-10-25,hello,good'
$ echo "$s" | awk -F, '{split($2,d,"-"); print "Month is: " d[2]}'
Month is: 10

$ # using regular expression to define separator
$ # return value is number of fields after splitting
$ s='Sample123string54with908numbers'
$ echo "$s" | awk '{n=split($0,s,/[0-9]+/); for(i=1;i<=n;i++)print s[i]}'
Sample
string
with
numbers
$ # use 4th argument if separators are needed as well
$ echo "$s" | awk '{n=split($0,s,/[0-9]+/,seps); for(i=1;i<n;i++)print seps[i]}'
123
54
908

$ # single row to multiple rows based on splitting last field
$ s='foo,baz,12:42:3'
$ echo "$s" | awk -F, '{n=split($NF,a,":"); NF--; for(i=1;i<=n;i++) print $0,a[i]}'
foo baz 12
foo baz 42
foo baz 3
```

- `substr` function allows to extract specified number of characters from given string
    - indexing starts with `1`
- See gawk manual - substr function for corner cases and details

```
$ # 1st argument is string to be worked on
$ # 2nd argument is starting position
$ # 3rd argument is number of characters to be extracted
$ echo 'abcdefghij' | awk '{print substr($0,1,5)}'
abcde
$ echo 'abcdefghij' | awk '{print substr($0,4,3)}'
def
$ # if 3rd argument is not given, string is extracted until end
$ echo 'abcdefghij' | awk '{print substr($0,6)}'
fghij

$ echo 'abcdefghij' | awk -v OFS=':' '{print substr($0,2,3), substr($0,6,3)}'
bcd:fgh

$ # if only few characters are needed from input line, can use empty FS
$ echo 'abcdefghij' | awk -v FS= '{print $3}'
c
$ echo 'abcdefghij' | awk -v FS= '{print $3, $5}'
c e
```

Executing external commands

- External commands can be issued using `system` function
- Output would be as usual on `stdout` unless redirected while calling the command
- Return value of `system` depends on `exit` status of executed command, see gawk manual - Input/Output Functions for details

```
$ awk 'BEGIN{system("echo Hello World")}'
Hello World

$ wc poem.txt
 4 13 65 poem.txt
$ awk 'BEGIN{system("wc poem.txt")}'
 4 13 65 poem.txt

$ awk 'BEGIN{system("seq 10 | paste -sd, > out.txt")}'
$ cat out.txt
1,2,3,4,5,6,7,8,9,10

$ ls xyz.txt
ls: cannot access 'xyz.txt': No such file or directory
$ echo $?
2
$ awk 'BEGIN{s=system("ls xyz.txt"); print "Status: " s}'
ls: cannot access 'xyz.txt': No such file or directory
Status: 2

$ cat f2
I bought two bananas and three mangoes
$ echo 'f1,f2,odd.txt' | awk -F, '{system("cat " $2)}'
I bought two bananas and three mangoes
```

printf formatting

- Similar to `printf` function in `C` and shell built-in command
- use `sprintf` function to save result in variable instead of printing
- See also gawk manual - printf

```
$ awk '{sum += $1} END{print sum}' nums.txt
10062.9

$ # note that ORS is not appended and has to be added manually
$ awk '{sum += $1} END{printf "%.2f\n", sum}' nums.txt
10062.86

$ awk '{sum += $1} END{printf "%10.2f\n", sum}' nums.txt
  10062.86

$ awk '{sum += $1} END{printf "%010.2f\n", sum}' nums.txt
0010062.86

$ awk '{sum += $1} END{printf "%d\n", sum}' nums.txt
10062

$ awk '{sum += $1} END{printf "%+d\n", sum}' nums.txt
+10062

$ awk '{sum += $1} END{printf "%e\n", sum}' nums.txt
1.006286e+04
```

- to refer argument by positional number (starts with 1), use `<num>$`

```
$ # can also use: awk 'BEGIN{printf "hex=%x\noct=%o\ndec=%d\n", 15, 15, 15}'
$ awk 'BEGIN{printf "hex=%1$x\noct=%1$o\ndec=%1$d\n", 15}'
hex=f
oct=17
dec=15

$ # adding prefix to hex/oct numbers
$ awk 'BEGIN{printf "hex=%1$#x\noct=%1$#o\ndec=%1$d\n", 15}'
hex=0xf
oct=017
dec=15
```

- strings

```
$ # prefix remaining width with spaces
$ awk 'BEGIN{printf "%6s:%5s\n", "foo", "bar"}'
   foo:  bar

$ # suffix remaining width with spaces
$ awk 'BEGIN{printf "%-6s:%-5s\n", "foo", "bar"}'
foo   :bar

$ # truncate
$ awk 'BEGIN{printf "%.2s\n", "foobar"}'
fo
```

- avoid using `printf` without format specifier

```
$ awk 'BEGIN{s="solve: 5 % x = 1"; printf s}'
awk: cmd. line:1: fatal: not enough arguments to satisfy format string
    `solve: 5 % x = 1'
               ^ ran out for this one

$ awk 'BEGIN{s="solve: 5 % x = 1"; printf "%s\n", s}'
solve: 5 % x = 1
```

Redirecting print output

- redirecting to file instead of stdout using `>`
- similar to behavior in shell, if file already exists it is overwritten
    - use `>>` to append to an existing file without deleting content
- however, unlike shell, subsequent redirections to same file will append to it
- See also gawk manual - Closing Input and Output Redirections if you have too many redirections

```
$ seq 6 | awk 'NR%2{print > "odd.txt"; next} {print > "even.txt"}'
$ cat odd.txt
1
3
5
$ cat even.txt
2
4
6

$ awk 'NR==1{col1=$1".txt"; col2=$2".txt"; next}
       {print $1 > col1; print $2 > col2}' fruits.txt
$ cat fruit.txt
apple
banana
fig
guava
$ cat qty.txt
42
31
90
6
```

- redirecting to shell command
- this is useful if you have different things to redirect to different commands, otherwise it can be done as usual in shell acting on `awk` 's output
- all redirections to same command gets combined as single input to that command

```
$ # same as: echo 'foo good 123' | awk '{print $2}' | wc -c
$ echo 'foo good 123' | awk '{print $2 | "wc -c"}'
5
$ # to avoid newline character being added to print
$ echo 'foo good 123' | awk -v ORS= '{print $2 | "wc -c"}'
4
$ # assuming no format specifiers in input
$ echo 'foo good 123' | awk '{printf $2 | "wc -c"}'
4

$ # same as: echo 'foo good 123' | awk '{printf $2 $3 | "wc -c"}'
$ echo 'foo good 123' | awk '{printf $2 | "wc -c"; printf $3 | "wc -c"}'
7
```

Further Reading

- gawk manual - Input/Output Functions

- gawk manual - Redirecting Output of print and printf
- gawk manual - Two-Way Communications with Another Process
- unix.stackexchange - inplace editing as well as stdout
- stackoverflow - redirect blocks to separate files

Gotchas and Tips

- using `$` for variables
- only input record `$0` and field contents `$1` , `$2` etc need `$`
- See also unix.stackexchange - Why does awk print the whole line when I want it to print a variable?

```
$ # wrong
$ awk -v word="apple" '$1==$word' fruits.txt

$ # right
$ awk -v word="apple" '$1==word' fruits.txt
apple   42
```

- dos style line endings
- See also unix.stackexchange - filtering when last column has \r

```
$ # no issue with unix style line ending
$ printf 'foo bar\n123 789\n' | awk '{print $2, $1}'
bar foo
789 123

$ # dos style line ending causes trouble
$ printf 'foo bar\r\n123 789\r\n' | awk '{print $2, $1}'
 foo
 123

$ # easy to deal by simply setting appropriate RS
$ # note that ORS would still be newline character only
$ printf 'foo bar\r\n123 789\r\n' | awk -v RS='\r\n' '{print $2, $1}'
bar foo
789 123
```

- relying on default intial value

```
$ # step 1 - works for single file
$ awk '{sum += $1} END{print sum}' nums.txt
10062.9

$ # step 2 - change to work for multiple file
$ awk '{sum += $1} ENDFILE{print FILENAME, sum}' nums.txt
nums.txt 10062.9

$ # step 3 - check with multiple file input
$ # oops, default numerical value '0' for sum works only once
$ awk '{sum += $1} ENDFILE{print FILENAME, sum}' nums.txt <(seq 3)
nums.txt 10062.9
/dev/fd/63 10068.9

$ # step 4 - correctly initialize variables
$ awk 'BEGINFILE{sum=0} {sum += $1} ENDFILE{print FILENAME, sum}' nums.txt <(seq 3)
nums.txt 10062.9
/dev/fd/63 6
```

- use unary operator `+` to force numeric conversion

```
$ awk '{sum += $1} END{print FILENAME, sum}' nums.txt
nums.txt 10062.9

$ awk '{sum += $1} END{print FILENAME, sum}' /dev/null
/dev/null

$ awk '{sum += $1} END{print FILENAME, +sum}' /dev/null
/dev/null 0
```

- concatenate empty string to force string comparison

```
$ echo '5 5.0' | awk '{print $1==$2 ? "same" : "different", "string"}'
same string

$ echo '5 5.0' | awk '{print $1""==$2 ? "same" : "different", "string"}'
different string
```

- beware of expressions going -ve for field calculations

```
$ cat misc.txt
foo
good bad ugly
123 xyz
a b c d

$ # trying to delete last two fields
$ awk '{NF -= 2} 1' misc.txt
awk: cmd. line:1: (FILENAME=misc.txt FNR=1) fatal: NF set to negative value
$ # dynamically change it depending on number of fields
$ awk '{NF = (NF<=2) ? 0 : NF-2} 1' misc.txt

good

a b

$ # similarly, trying to access 3rd field from end
$ awk '{print $(NF-2)}' misc.txt
awk: cmd. line:1: (FILENAME=misc.txt FNR=1) fatal: attempt to access field -1
$ awk 'NF>2{print $(NF-2)}' misc.txt
good
b
```

- If input is ASCII alone, simple trick to improve speed

```
$ # all words containing exactly 3 lowercase a
$ time awk -F'a' 'NF==4{cnt++} END{print +cnt}' /usr/share/dict/words
1019

real    0m0.075s

$ time LC_ALL=C awk -F'a' 'NF==4{cnt++} END{print +cnt}' /usr/share/dict/words
1019

real    0m0.045s
```

Further Reading

- `man awk` and `info awk` for quick reference from command line
- gawk manual for complete reference, extensions and more
- What's up with different `awk` versions?
    - unix.stackexchange - brief explanation

    - Differences between gawk, nawk, mawk, and POSIX awk
    - cheat sheet for awk/nawk/gawk
- Tutorials and Q&A
    - code.snipcademy - gentle intro
    - funtoo - using examples
    - grymoire - detailed tutorial - covers information about different `awk` versions as well
    - catonmat - one liners explained
    - awk Q&A on stackoverflow
    - awk Q&A on unix.stackexchange
- Alternatives
    - GNU datamash
    - bioawk
    - hawk - based on Haskell
    - miller - similar to awk/sed/cut/join/sort for name-indexed data such as CSV, TSV, and tabular JSON
        - See this ycombinator news for other tools like this
- unix.stackexchange - When to use grep, sed, awk, perl, etc
- awkaster - Pseudo-3D shooter written completely in awk using raycasting technique
- examples for some of the stuff not covered in this tutorial
    - unix.stackexchange - rand/srand
    - unix.stackexchange - strftime
    - unix.stackexchange - ARGC and ARGV
    - stackoverflow - arbitrary precision integer extension
    - unix.stackexchange - sprintf and close
    - unix.stackexchange - user defined functions and array passing

Sorting stuff

Table of Contents

sort

```
$ sort --version | head -n1
sort (GNU coreutils) 8.25

$ man sort
SORT(1)                          User Commands                         SORT(1)

NAME
       sort - sort lines of text files

SYNOPSIS
       sort [OPTION]... [FILE]...
       sort [OPTION]... --files0-from=F

DESCRIPTION
       Write sorted concatenation of all FILE(s) to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

Note: All examples shown here assumes ASCII encoded input file

Default sort

```
$ cat poem.txt
Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.

$ sort poem.txt
And so are you.
Roses are red,
Sugar is sweet,
Violets are blue,
```

- Well, that was easy. The lines were sorted alphabetically (ascending order by default) and it so happened that first letter alone was enough to decide the order
- For next example, let's extract all the words and sort them
  - also allows to showcase `sort` accepting stdin
  - See GNU grep chapter if the `grep` command used below looks alien

```
$ # output might differ depending on locale settings
$ # note the case-insensitiveness of output
$ grep -oi '[a-z]*' poem.txt | sort
And
are
are
are
blue
is
red
Roses
so
Sugar
sweet
Violets
you
```

- heed hereunto
- See also
  - arch wiki - locale
  - Linux: Define Locale and Language Settings

```
$ info sort | tail

   (1) 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’.
```

- Example to help show effect of locale setting

```
$ # note how uppercase is sorted before lowercase
$ grep -oi '[a-z]*' poem.txt | LC_ALL=C sort
And
Roses
Sugar
Violets
are
are
are
blue
is
red
so
sweet
you
```

Reverse sort

- This is simply reversing from default ascending order to descending order

```
$ sort -r poem.txt
Violets are blue,
Sugar is sweet,
Roses are red,
And so are you.
```

Various number sorting

```
$ cat numbers.txt
20
53
3
101

$ sort numbers.txt
101
20
3
53
```

- Whoops, what happened there? `sort` won't know to treat them as numbers unless specified
- Depending on format of numbers, different options have to be used
- First up is `-n` option, which sorts based on numerical value

```
$ sort -n numbers.txt
3
20
53
101

$ sort -nr numbers.txt
101
53
20
3
```

- The `-n` option can handle negative numbers
- As well as thousands separator and decimal point (depends on locale)
- The `<()` syntax is Process Substitution
  - to put it simply - allows output of command to be passed as input file to another command without needing to manually create a temporary file

```
$ # multiple files are merged as single input by default
$ sort -n numbers.txt <(echo '-4')
-4
3
20
53
101

$ sort -n numbers.txt <(echo '1,234')
3
20
53
101
1,234

$ sort -n numbers.txt <(echo '31.24')
3
20
31.24
53
101
```

- Use `-g` if input contains numbers prefixed by `+` or E scientific notation

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

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

- Commands like `du` have options to display numbers in human readable formats
- `sort` supports sorting such numbers using the `-h` option

```
$ du -sh *
104K    power.log
746M    projects
316K    report.log
20K     sample.txt
$ du -sh * | sort -h
20K     sample.txt
104K    power.log
316K    report.log
746M    projects

$ # --si uses powers of 1000 instead of 1024
$ du -s --si *
107k    power.log
782M    projects
324k    report.log
21k     sample.txt
$ du -s --si * | sort -h
21k     sample.txt
107k    power.log
324k    report.log
782M    projects
```

- Version sort - dealing with numbers mixed with other characters
- If this sorting is needed simply while displaying directory contents, use `ls -v` instead of piping to `sort -V`

```
$ cat versions.txt
foo_v1.2
bar_v2.1.3
foobar_v2
foo_v1.2.1
foo_v1.3

$ sort -V versions.txt
bar_v2.1.3
foobar_v2
foo_v1.2
foo_v1.2.1
foo_v1.3
```

- Another common use case is when there are multiple filenames differentiated by numbers

```
$ cat files.txt
file0
file10
file3
file4

$ sort -V files.txt
file0
file3
file4
file10
```

- Can be used when dealing with numbers reported by `time` command as well

```
$ # different solving durations
$ cat rubik_time.txt
5m35.363s
3m20.058s
4m5.099s
4m1.130s
3m42.833s
4m33.083s

$ # assuming consistent min/sec format
$ sort -V rubik_time.txt
3m20.058s
3m42.833s
4m1.130s
4m5.099s
4m33.083s
5m35.363s
```

Random sort

- Note that duplicate lines will always end up next to each other
  - might be useful as a feature for some cases ;)
  - Use `shuf` if this is not desirable
- See also How can I shuffle the lines of a text file on the Unix command line or in a shell script?

```
$ cat nums.txt
1
10
10
12
23
563

$ # the two 10s will always be next to each other
$ sort -R nums.txt
563
12
1
10
10
23

$ # duplicates can end up anywhere
$ shuf nums.txt
10
23
1
10
563
12
```

Specifying output file

- The `-o` option can be used to specify output file
- Useful for in place editing

```
$ sort -R nums.txt -o rand_nums.txt
$ cat rand_nums.txt
23
1
10
10
563
12

$ sort -R nums.txt -o nums.txt
$ cat nums.txt
563
23
10
10
1
12
```

- Use shell script looping if there multiple files to be sorted in place
- Below snippet is for `bash` shell

```
$ for f in *.txt; do echo sort -V "$f" -o "$f"; done
sort -V files.txt -o files.txt
sort -V rubik_time.txt -o rubik_time.txt
sort -V versions.txt -o versions.txt

$ # remove echo once commands look fine
$ for f in *.txt; do sort -V "$f" -o "$f"; done
```

Unique sort

- Keep only first copy of lines that are deemed to be same according to `sort` option used

```
$ cat duplicates.txt
foo
12 carrots
foo
12 apples
5 guavas

$ # only one copy of foo in output
$ sort -u duplicates.txt
12 apples
12 carrots
5 guavas
foo
```

- According to option used, definition of duplicate will vary
- For example, when `-n` is used, matching numbers are deemed same even if rest of line differs
  - Pipe the output to `uniq` if this is not desirable

```
$ # note how first copy of line starting with 12 is retained
$ sort -nu duplicates.txt
foo
5 guavas
12 carrots

$ # use uniq when entire line should be compared to find duplicates
$ sort -n duplicates.txt | uniq
foo
5 guavas
12 apples
12 carrots
```

- Use `-f` option to ignore case of alphabets while determining duplicates

```
$ cat words.txt
CAR
are
car
Are
foot
are

$ # only the two 'are' were considered duplicates
$ sort -u words.txt
are
Are
car
CAR
foot

$ # note again that first copy of duplicate is retained
$ sort -fu words.txt
are
CAR
foot
```

Column based sorting

From `info sort`

```
‘-k POS1[,POS2]’
‘--key=POS1[,POS2]’
     Specify a sort field that consists of the part of the line between
     POS1 and POS2 (or the end of the line, if POS2 is omitted),
     _inclusive_.

     Each POS has the form ‘F[.C][OPTS]’, where F is the number of the
     field to use, and C is the number of the first character from the
     beginning of the field.  Fields and character positions are
     numbered starting with 1; a character position of zero in POS2
     indicates the field’s last character.  If ‘.C’ is omitted from
     POS1, it defaults to 1 (the beginning of the field); if omitted
     from POS2, it defaults to 0 (the end of the field).  OPTS are
     ordering options, allowing individual keys to be sorted according
     to different rules; see below for details.  Keys can span multiple
     fields.
```

- By default, blank characters (space and tab) serve as field separators

```
$ cat fruits.txt
apple   42
guava   6
fig     90
banana  31

$ sort fruits.txt
apple   42
banana  31
fig     90
guava   6

$ # sort based on 2nd column numbers
$ sort -k2,2n fruits.txt
guava   6
banana  31
apple   42
fig     90
```

- Using a different field separator
- Consider the following sample input file having fields separated by `:`

```
$ # name:pet_name:no_of_pets
$ cat pets.txt
foo:dog:2
xyz:cat:1
baz:parrot:5
abcd:cat:3
joe:dog:1
bar:fox:1
temp_var:squirrel:4
boss:dog:10
```

- Sorting based on particular column or column to end of line
- In case of multiple entries, by default `sort` would use content of remaining parts of line to resolve

```
$ # only 2nd column
$ # -k2,4 would mean 2nd column to 4th column
$ sort -t: -k2,2 pets.txt
abcd:cat:3
xyz:cat:1
boss:dog:10
foo:dog:2
joe:dog:1
bar:fox:1
baz:parrot:5
temp_var:squirrel:4

$ # from 2nd column to end of line
$ sort -t: -k2 pets.txt
xyz:cat:1
abcd:cat:3
joe:dog:1
boss:dog:10
foo:dog:2
bar:fox:1
baz:parrot:5
temp_var:squirrel:4
```

- Multiple keys can be specified to resolve ties
- Note that if there are still multiple entries with specified keys, remaining parts of lines would be used

```
$ # default sort for 2nd column, numeric sort on 3rd column to resolve ties
$ sort -t: -k2,2 -k3,3n pets.txt
xyz:cat:1
abcd:cat:3
joe:dog:1
foo:dog:2
boss:dog:10
bar:fox:1
baz:parrot:5
temp_var:squirrel:4

$ # numeric sort on 3rd column, default sort for 2nd column to resolve ties
$ sort -t: -k3,3n -k2,2 pets.txt
xyz:cat:1
joe:dog:1
bar:fox:1
foo:dog:2
abcd:cat:3
temp_var:squirrel:4
baz:parrot:5
boss:dog:10
```

- Use `-s` option to retain original order of lines in case of tie

```
$ sort -s -t: -k2,2 pets.txt
xyz:cat:1
abcd:cat:3
foo:dog:2
joe:dog:1
boss:dog:10
bar:fox:1
baz:parrot:5
temp_var:squirrel:4
```

- The `-u` option, as seen earlier, will retain only first match

```
$ sort -u -t: -k2,2 pets.txt
xyz:cat:1
foo:dog:2
bar:fox:1
baz:parrot:5
temp_var:squirrel:4

$ sort -u -t: -k3,3n pets.txt
xyz:cat:1
foo:dog:2
abcd:cat:3
temp_var:squirrel:4
baz:parrot:5
boss:dog:10
```

- Sometimes, the input has to be sorted first and then `-u` used on the sorted output
- See also remove duplicates based on the value of another column

```
$ # sort by number in 3rd column
$ sort -t: -k3,3n pets.txt
bar:fox:1
joe:dog:1
xyz:cat:1
foo:dog:2
abcd:cat:3
temp_var:squirrel:4
baz:parrot:5
boss:dog:10

$ # then get unique entry based on 2nd column
$ sort -t: -k3,3n pets.txt | sort -t: -u -k2,2
xyz:cat:1
joe:dog:1
bar:fox:1
baz:parrot:5
temp_var:squirrel:4
```

- Specifying particular characters within fields
- If character position is not specified, defaults to `1` for starting column and `0` (last character) for ending column

```
$ cat marks.txt
fork,ap_12,54
flat,up_342,1.2
fold,tn_48,211
more,ap_93,7
rest,up_5,63

$ # for 2nd column, sort numerically only from 4th character to end
$ sort -t, -k2.4,2n marks.txt
rest,up_5,63
fork,ap_12,54
fold,tn_48,211
more,ap_93,7
flat,up_342,1.2

$ # sort uniquely based on first two characters of line
$ sort -u -k1.1,1.2 marks.txt
flat,up_342,1.2
fork,ap_12,54
more,ap_93,7
rest,up_5,63
```

- If there are headers

```
$ cat header.txt
fruit   qty
apple   42
guava   6
fig     90
banana  31

$ # separate and combine header and content to be sorted
$ cat <(head -n1 header.txt) <(tail -n +2 header.txt | sort -k2nr)
fruit   qty
fig     90
apple   42
banana  31
guava   6
```

- See also sort by last field value when number of fields varies

Further reading for sort

- There are many other options apart from handful presented above. See `man sort` and `info`

`sort` for detailed documentation and more examples
- sort like a master
- When -b to ignore leading blanks is needed
- sort Q&A on unix stackexchange
- sort on multiple columns using -k option
- sort a string character wise
- Scalability of 'sort -u' for gigantic files

uniq

```
$ uniq --version | head -n1
uniq (GNU coreutils) 8.25

$ man uniq
UNIQ(1)                          User Commands                          UNIQ(1)

NAME
       uniq - report or omit repeated lines

SYNOPSIS
       uniq [OPTION]... [INPUT [OUTPUT]]

DESCRIPTION
       Filter  adjacent matching lines from INPUT (or standard input), writing
       to OUTPUT (or standard output).

       With no options, matching lines are merged to the first occurrence.
...
```

Default uniq

```
$ cat word_list.txt
are
are
to
good
bad
bad
bad
good
are
bad

$ # adjacent duplicate lines are removed, leaving one copy
$ uniq word_list.txt
are
to
good
bad
good
are
bad

$ # To remove duplicates from entire file, input has to be sorted first
$ # also showcases that uniq accepts stdin as input
$ sort word_list.txt | uniq
are
bad
good
to
```

Only duplicates

```
$ # duplicates adjacent to each other
$ uniq -d word_list.txt
are
bad

$ # duplicates in entire file
$ sort word_list.txt | uniq -d
are
bad
good
```

- To get only duplicates as well as show all duplicates

```
$ uniq -D word_list.txt
are
are
bad
bad
bad

$ sort word_list.txt | uniq -D
are
are
are
bad
bad
bad
bad
good
good
```

- To distinguish the different groups

```
$ # using --all-repeated=prepend will add a newline before the first group as well
$ sort word_list.txt | uniq --all-repeated=separate
are
are
are

bad
bad
bad
bad

good
good
```

Only unique

```
$ # lines with no adjacent duplicates
$ uniq -u word_list.txt
to
good
good
are
bad

$ # unique lines in entire file
$ sort word_list.txt | uniq -u
to
```

Prefix count

```
$ # adjacent lines
$ uniq -c word_list.txt
      2 are
      1 to
      1 good
      3 bad
      1 good
      1 are
      1 bad

$ # entire file
$ sort word_list.txt | uniq -c
      3 are
      4 bad
      2 good
      1 to

$ # entire file, only duplicates
$ sort word_list.txt | uniq -cd
      3 are
      4 bad
      2 good
```

- Sorting by count

```
$ # sort by count
$ sort word_list.txt | uniq -c | sort -n
      1 to
      2 good
      3 are
      4 bad

$ # reverse the order, highest count first
$ sort word_list.txt | uniq -c | sort -nr
      4 bad
      3 are
      2 good
      1 to
```

- To get only entries with min/max count, bit of awk magic would help

```
$ # consider this result
$ sort colors.txt | uniq -c | sort -nr
      3 Red
      3 Blue
      2 Yellow
      1 Green
      1 Black

$ # to get all max count
$ # save 1st line 1st column value to c and then print if 1st column equals c
$ sort colors.txt | uniq -c | sort -nr | awk 'NR==1{c=$1} $1==c'
      3 Red
      3 Blue
$ # to get all min count
$ sort colors.txt | uniq -c | sort -n | awk 'NR==1{c=$1} $1==c'
      1 Black
      1 Green
```

- Get rough count of most used commands from `history` file

```
$ # awk '{print $1}' will get the 1st column alone
$ awk '{print $1}' "$HISTFILE" | sort | uniq -c | sort -nr | head
   1465 echo
   1180 grep
    552 cd
    531 awk
    451 sed
    423 vi
    418 cat
    392 perl
    325 printf
    320 sort

$ # extract command name from start of line or preceded by 'spaces|spaces'
$ # won't catch commands in other places like command substitution though
$ grep -oP '(^| +\| +)\K[^ ]+' "$HISTFILE" | sort | uniq -c | sort -nr | head
   2006 grep
   1469 echo
    933 sed
    698 awk
    552 cd
    513 perl
    510 cat
    453 sort
    423 vi
    327 printf
```

Ignoring case

```
$ cat another_list.txt
food
Food
good
are
bad
Are

$ # note how first copy is retained
$ uniq -i another_list.txt
food
good
are
bad
Are

$ uniq -iD another_list.txt
food
Food
```

Combining multiple files

```
$ sort -f word_list.txt another_list.txt | uniq -i
are
bad
food
good
to

$ sort -f word_list.txt another_list.txt | uniq -c
      4 are
      1 Are
      5 bad
      1 food
      1 Food
      3 good
      1 to

$ sort -f word_list.txt another_list.txt | uniq -ic
      5 are
      5 bad
      2 food
      3 good
      1 to
```

- If only adjacent lines (not sorted) is required, need to concatenate files using another command

```
$ uniq -id word_list.txt
are
bad

$ uniq -id another_list.txt
food

$ cat word_list.txt another_list.txt | uniq -id
are
bad
food
```

Column options

- `uniq` has few options dealing with column manipulations. Not extensive as `sort -k` but handy for some cases
- First up, skipping fields
    - No option to specify different delimiter

    - From `info uniq` : Fields are sequences of non-space non-tab characters that are separated from each other by at least one space or tab
    - Number of spaces/tabs between fields should be same

```
$ cat shopping.txt
lemon 5
mango 5
banana 8
bread 1
orange 5

$ # skips first field
$ uniq -f1 shopping.txt
lemon 5
banana 8
bread 1
orange 5

$ # use -f3 to skip first three fields and so on
```

- Skipping characters

```
$ cat text
glue
blue
black
stack
stuck

$ # don't consider first 2 characters
$ uniq -s2 text
glue
black
stuck

$ # to visualize the above example
$ # assume there are two fields and uniq is applied on 2nd column
$ sed 's/^../& /' text
gl ue
bl ue
bl ack
st ack
st uck
```

- Upto specified characters

```
$ # consider only first 2 characters
$ uniq -w2 text
glue
blue
stack

$ # to visualize the above example
$ # assume there are two fields and uniq is applied on 1st column
$ sed 's/^../& /' text
gl ue
bl ue
bl ack
st ack
st uck
```

- Combining `-s` and `-w`
- Can be combined with `-f` as well

```
$ # skip first 3 characters and then use next 2 characters
$ uniq -s3 -w2 text
glue
black
```

Further reading for uniq

- Do check out `man uniq` and `info uniq` for other options and more detailed documentation
- uniq Q&A on unix stackexchange
- process duplicate lines only based on certain fields

comm

```
$ comm --version | head -n1
comm (GNU coreutils) 8.25

$ man comm
COMM(1)                          User Commands                          COMM(1)

NAME
       comm - compare two sorted files line by line

SYNOPSIS
       comm [OPTION]... FILE1 FILE2

DESCRIPTION
       Compare sorted files FILE1 and FILE2 line by line.

       When FILE1 or FILE2 (not both) is -, read standard input.

       With  no  options,  produce  three-column  output.  Column one contains
       lines unique to FILE1, column two contains lines unique to  FILE2,  and
       column three contains lines common to both files.
...
```

Default three column output

Consider below sample input files

```
$ # sorted input files viewed side by side
$ paste colors_1.txt colors_2.txt
Blue    Black
Brown   Blue
Purple  Green
Red     Red
Teal    White
Yellow
```

- Without any option, `comm` gives 3 column output
    - lines unique to first file
    - lines unique to second file
    - lines common to both files

```
$ comm colors_1.txt colors_2.txt
        Black
                Blue
Brown
        Green
Purple
                Red
Teal
        White
Yellow
```

Suppressing columns

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

```
$ # suppressing column 3
$ comm -3 colors_1.txt colors_2.txt
        Black
Brown
        Green
Purple
Teal
        White
Yellow
```

- Combining options gives three distinct and useful constructs
- First, getting only common lines to both files

```
$ comm -12 colors_1.txt colors_2.txt
Blue
Red
```

- Second, lines unique to first file

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

- And the third, lines unique to second file

```
$ comm -13 colors_1.txt colors_2.txt
Black
Green
White
```

- See also how the above three cases can be done using grep alone
  - **Note** input files do not need to be sorted for `grep` solution

If different `sort` order than default is required, use `--nocheck-order` to ignore error message

```
$ comm -23 <(sort -n numbers.txt) <(sort -n nums.txt)
3
comm: file 1 is not in sorted order
20
53
101

$ comm --nocheck-order -23 <(sort -n numbers.txt) <(sort -n nums.txt)
3
20
53
101
```

Files with duplicates

- As many duplicate lines match in both files, they'll be considered as common
- Rest will be unique to respective files
- This is useful for cases like finding lines present in first but not in second taking in to consideration count of duplicates as well
  - This solution won't be possible with `grep`

```
$ paste list1 list2
a       a
a       b
a       c
b       c
b       d
c

$ comm list1 list2
                a
a
a
                b
b
                c
        c
        d

$ comm -23 list1 list2
a
a
b
```

Further reading for comm

- `man comm` and `info comm` for more options and detailed documentation
- comm Q&A on unix stackexchange

shuf

```
$ shuf --version | head -n1
shuf (GNU coreutils) 8.25

$ man shuf
SHUF(1)                          User Commands                          SHUF(1)

NAME
       shuf - generate random permutations

SYNOPSIS
       shuf [OPTION]... [FILE]
       shuf -e [OPTION]... [ARG]...
       shuf -i LO-HI [OPTION]...

DESCRIPTION
       Write a random permutation of the input lines to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

Random lines

- Without repeating input lines

```
$ cat nums.txt
1
10
10
12
23
563

$ # duplicates can end up anywhere
$ # all lines are part of output
$ shuf nums.txt
10
23
1
10
563
12

$ # limit max number of output lines
$ shuf -n2 nums.txt
563
23
```

- Use `-o` option to specify output file name instead of displaying on stdout
- Helpful for inplace editing

```
$ shuf nums.txt -o nums.txt
$ cat nums.txt
10
12
23
10
563
1
```

- With repeated input lines

```
$ # -n3 for max 3 lines, -r allows input lines to be repeated
$ shuf -n3 -r nums.txt
1
1
563

$ seq 3 | shuf -n5 -r
2
1
2
1
2

$ # if a limit using -n is not specified, shuf will output lines indefinitely
```

- use `-e` option to specify multiple input lines from command line itself

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

$ shuf -e 'hi there' 'hello world' foo bar
bar
hi there
foo
hello world

$ shuf -n2 -e 'hi there' 'hello world' foo bar
foo
hi there

$ shuf -r -n4 -e foo bar
foo
foo
bar
foo
```

Random integer numbers

- The `-i` option accepts integer range as input to be shuffled

```
$ shuf -i 3-8
3
7
6
4
8
5
```

- Combine with other options as needed

```
$ shuf -n3 -i 3-8
5
4
7

$ shuf -r -n4 -i 3-8
5
5
7
8

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

- Use seq input if negative numbers, floating point, etc are needed

```
$ seq 2 -1 -2 | shuf
2
-1
-2
0
1

$ seq 0.3 0.1 0.7 | shuf -n3
0.4
0.5
0.7
```

Further reading for shuf

- `man shuf` and `info shuf` for more options and detailed documentation
- Generate random numbers in specific range
- Variable - randomly choose among three numbers
- Related to 'random' stuff:
    - How to generate a random string?
    - How can I populate a file with random data?
    - Run commands at random

Restructure text

Table of Contents

paste

```
$ paste --version | head -n1
paste (GNU coreutils) 8.25

$ man paste
PASTE(1)                         User Commands                        PASTE(1)

NAME
       paste - merge lines of files

SYNOPSIS
       paste [OPTION]... [FILE]...

DESCRIPTION
       Write  lines  consisting  of  the sequentially corresponding lines from
       each FILE, separated by TABs, to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

Concatenating files column wise

- By default, `paste` adds a TAB between corresponding lines of input files

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

- Specifying a different delimiter using `-d`
- The `<()` syntax is Process Substitution
    - to put it simply - allows output of command to be passed as input file to another command without needing to manually create a temporary file

```
$ paste -d, <(seq 5) <(seq 6 10)
1,6
2,7
3,8
4,9
5,10

$ # empty cells if number of lines is not same for all input files
$ # -d\| can also be used
$ paste -d'|' <(seq 3) <(seq 4 6) <(seq 7 10)
1|4|7
2|5|8
3|6|9
||10
```

Interleaving lines

- Interleave lines by using newline as delimiter

```
$ paste -d'\n' <(seq 11 13) <(seq 101 103)
11
101
12
102
13
103
```

Lines to multiple columns

- Number of `-` specified determines number of output columns
- Input lines can be passed only as stdin

```
$ # single column to two columns
$ seq 10 | paste -d, - -
1,2
3,4
5,6
7,8
9,10

$ # single column to five columns
$ seq 10 | paste -d: - - - - -
1:2:3:4:5
6:7:8:9:10

$ # input redirection for file input
$ paste -d, - - < colors_1.txt
Blue,Brown
Purple,Red
Teal,
```

- Use `printf` trick if number of columns to specify is too large

```
$ # prompt at end of line not shown for simplicity
$ printf -- "- %.s" {1..5}
- - - - -

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

Different delimiters between columns

- For more than 2 columns, different delimiter character can be specified - passed as list to `-d` option

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

$ # re-use list from beginning if not specified for all columns
$ 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
$ # another example
$ seq 10 | paste -d':,' - - - - -
1:2,3:4,5
6:7,8:9,10

$ # so, with single delimiter, it is just re-used for all columns
$ 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
```

- combination of `-d` and `/dev/null` (empty file) can give multi-character separation between columns
- If this is too confusing to use, consider pr instead

```
$ paste -d' : ' <(seq 3) /dev/null /dev/null <(seq 4 6) /dev/null /dev/null <(seq 7
9)
1 : 4 : 7
2 : 5 : 8
3 : 6 : 9

$ # or just use pr instead
$ pr -mts' : ' <(seq 3) <(seq 4 6) <(seq 7 9)
1 : 4 : 7
2 : 5 : 8
3 : 6 : 9

$ # but paste would allow different delimiters ;)
$ paste -d' :  - ' <(seq 3) /dev/null /dev/null <(seq 4 6) /dev/null /dev/null <(seq
 7 9)
1 : 4 - 7
2 : 5 - 8
3 : 6 - 9

$ # pr would need two invocations
$ pr -mts' : ' <(seq 3) <(seq 4 6) | pr -mts' - ' - <(seq 7 9)
1 : 4 - 7
2 : 5 - 8
3 : 6 - 9
```

- example to show using empty file instead of `/dev/null`

```
$ # assuming file named e doesn't exist
$ touch e
$ # or use this, will empty contents even if file named e already exists :P
$ > e

$ paste -d' :  - ' <(seq 3) e e <(seq 4 6) e e <(seq 7 9)
1 : 4 - 7
2 : 5 - 8
3 : 6 - 9
```

Multiple lines to single row

```
$ paste -sd, colors_1.txt
Blue,Brown,Purple,Red,Teal

$ # multiple files each gets a row
$ paste -sd: colors_1.txt colors_2.txt
Blue:Brown:Purple:Red:Teal
Black:Blue:Green:Red:White

$ # multiple input files need not have same number of lines
$ paste -sd, <(seq 3) <(seq 5 9)
1,2,3
5,6,7,8,9
```

- Often used to serialize multiple line output from another command

```
$ sort -u colors_1.txt colors_2.txt | paste -sd,
Black,Blue,Brown,Green,Purple,Red,Teal,White
```

- For multiple character delimiter, post-process if separator is unique or use another tool like `perl`

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

$ # post-process
$ seq 10 | paste -sd, | sed 's/,/ : /g'
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10

$ # using perl alone
$ seq 10 | perl -pe 's/\n/ : / if(!eof)'
1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10
```

Further reading for paste

- `man paste` and `info paste` for more options and detailed documentation
- paste Q&A on unix stackexchange

column

```
COLUMN(1)                 BSD General Commands Manual                COLUMN(1)

NAME
     column — columnate lists

SYNOPSIS
     column [-entx] [-c columns] [-s sep] [file ...]

DESCRIPTION
     The column utility formats its input into multiple columns.  Rows are
     filled before columns.  Input is taken from file operands, or, by
     default, from the standard input.  Empty lines are ignored unless the -e
     option is used.
...
```

Pretty printing tables

- by default whitespace is input delimiter

```
$ cat dishes.txt
North alootikki baati khichdi makkiroti poha
South appam bisibelebath dosa koottu sevai
West dhokla khakhra modak shiro vadapav
East handoguri litti momo rosgulla shondesh

$ column -t dishes.txt
North  alootikki  baati         khichdi  makkiroti  poha
South  appam      bisibelebath  dosa     koottu     sevai
West   dhokla     khakhra       modak    shiro      vadapav
East   handoguri  litti         momo     rosgulla   shondesh
```

- often useful to get neatly aligned columns from output of another command

```
$ paste fruits.txt price.txt
Fruits  Price
apple   182
guava   90
watermelon      35
banana  72
pomegranate     280

$ paste fruits.txt price.txt | column -t
Fruits       Price
apple        182
guava        90
watermelon   35
banana       72
pomegranate  280
```

Specifying different input delimiter

- Use `-s` to specify input delimiter
- Use `-n` to prevent merging empty cells
    - From `man column` "This option is a Debian GNU/Linux extension"

```
$ paste -d, <(seq 3) <(seq 5 9) <(seq 11 13)
1,5,11
2,6,12
3,7,13
,8,
,9,

$ paste -d, <(seq 3) <(seq 5 9) <(seq 11 13) | column -s, -t
1  5  11
2  6  12
3  7  13
8
9

$ paste -d, <(seq 3) <(seq 5 9) <(seq 11 13) | column -s, -nt
1  5  11
2  6  12
3  7  13
   8
   9
```

Further reading for column

- `man column` for more options and detailed documentation
- column Q&A on unix stackexchange
- More examples here

pr

```
$ pr --version | head -n1
pr (GNU coreutils) 8.25

$ man pr
PR(1)                            User Commands                           PR(1)

NAME
       pr - convert text files for printing

SYNOPSIS
       pr [OPTION]... [FILE]...

DESCRIPTION
       Paginate or columnate FILE(s) for printing.

       With no FILE, or when FILE is -, read standard input.
...
```

- `Paginate` is not covered, examples related only to `columnate`
- For example, default invocation on a file would add a header, etc

```
$ # truncated output shown
$ pr fruits.txt


2017-04-21 17:49                    fruits.txt                    Page 1


Fruits
apple
guava
watermelon
banana
pomegranate
```

- Following sections will use `-t` to omit page headers and trailers

Converting lines to columns

- With paste, changing input file rows to column(s) is possible only with consecutive lines
- `pr` can do that as well as split entire file itself according to number of columns needed
- And `-s` option in `pr` allows multi-character output delimiter
- As usual, examples to better show the functionalities

```
$ # note how the input got split into two and resulting splits joined by ,
$ seq 6 | pr -2ts,
1,4
2,5
3,6

$ # note how two consecutive lines gets joined by ,
$ seq 6 | paste -d, - -
1,2
3,4
5,6
```

- Default **PAGE_WIDTH** is 72 characters, so each column gets 72 divided by number of columns unless `-s` is used

```
$ # 3 columns, so each column width is 24 characters
$ seq 9 | pr -3t
1                       4                       7
2                       5                       8
3                       6                       9

$ # using -s, desired delimiter can be specified
$ seq 9 | pr -3ts' '
1 4 7
2 5 8
3 6 9

$ seq 9 | pr -3ts' : '
1 : 4 : 7
2 : 5 : 8
3 : 6 : 9

$ # default is TAB when using -s option with no arguments
$ seq 9 | pr -3ts
1       4       7
2       5       8
3       6       9
```

- Using `-a` to change consecutive rows, similar to `paste`

```
$ seq 8 | pr -4ats:
1:2:3:4
5:6:7:8

$ # no output delimiter for empty cells
$ seq 22 | pr -5ats,
1,2,3,4,5
6,7,8,9,10
11,12,13,14,15
16,17,18,19,20
21,22

$ # note output delimiter even for empty cells
$ seq 22 | paste -d, - - - - -
1,2,3,4,5
6,7,8,9,10
11,12,13,14,15
16,17,18,19,20
21,22,,,
```

Changing PAGE_WIDTH

- The default PAGE_WIDTH is 72
- The formula `(col-1)*len(delimiter) + col` seems to work in determining minimum PAGE_WIDTH required for multiple column output
  - `col` is number of columns required

```
$ # (36-1)*1 + 36 = 71, so within PAGE_WIDTH limit
$ seq 74 | pr -36ats,
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32,33,34,35,36
37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,
65,66,67,68,69,70,71,72
73,74
$ # (37-1)*1 + 37 = 73, more than default PAGE_WIDTH limit
$ seq 74 | pr -37ats,
pr: page width too narrow
```

- Use `-w` to specify a different PAGE_WIDTH
- The `-J` option turns off truncation

```
$ # (37-1)*1 + 37 = 73
$ seq 74 | pr -J -w73 -37ats,
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,
32,33,34,35,36,37
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,
66,67,68,69,70,71,72,73,74

$ # (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

$ # if calculating is difficult, simply use a large number
$ seq 6 | pr -J -w500 -3ats'::::'
1::::2::::3
4::::5::::6
```

Combining multiple input files

- Use `-m` option to combine multiple files in parallel, similar to `paste`

```
$ # 2 columns, so each column width is 36 characters
$ pr -mt fruits.txt price.txt
Fruits                              Price
apple                               182
guava                               90
watermelon                          35
banana                              72
pomegranate                         280

$ # default is TAB when using -s option with no arguments
$ pr -mts <(seq 3) <(seq 4 6) <(seq 7 10)
1       4       7
2       5       8
3       6       9
                10

$ # double TAB as separator
$ # shell expands $'\t\t' before command is executed
$ pr -mts$'\t\t' colors_1.txt colors_2.txt
Blue            Black
Brown           Blue
Purple          Green
Red             Red
Teal            White
```

- For interleaving, specify newline as separator

```
$ pr -mts$'\n' fruits.txt price.txt
Fruits
Price
apple
182
guava
90
watermelon
35
banana
72
pomegranate
280
```

Transposing a table

```
$ # delimiter is single character, so easy to use tr to change it to newline
$ cat dishes.txt
North alootikki baati khichdi makkiroti poha
South appam bisibelebath dosa koottu sevai
West dhokla khakhra modak shiro vadapav
East handoguri litti momo rosgulla shondesh

$ # 4 columns, so each column width is 18 characters
$ # $(wc -l < dishes.txt) gives number of columns required
$ tr ' ' '\n' < dishes.txt | pr -$(wc -l < dishes.txt)t
North             South             West              East
alootikki         appam             dhokla            handoguri
baati             bisibelebath      khakhra           litti
khichdi           dosa              modak             momo
makkiroti         koottu            shiro             rosgulla
poha              sevai             vadapav           shondesh
```

- Pipe the output to `column` if spacing is too much

```
$ tr ' ' '\n' < dishes.txt | pr -$(wc -l < dishes.txt)t | column -t
North      South         West     East
alootikki  appam         dhokla   handoguri
baati      bisibelebath  khakhra  litti
khichdi    dosa          modak    momo
makkiroti  koottu        shiro    rosgulla
poha       sevai         vadapav  shondesh
```

Further reading for pr

- `man pr` and `info pr` for more options and detailed documentation
- More examples here

fold

```
$ fold --version | head -n1
fold (GNU coreutils) 8.25

$ man fold
FOLD(1)                          User Commands                          FOLD(1)

NAME
       fold - wrap each input line to fit in specified width

SYNOPSIS
       fold [OPTION]... [FILE]...

DESCRIPTION
       Wrap input lines in each FILE, writing to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

Examples

```
$ nl story.txt
     1	The princess of a far away land fought bravely to rescue a travelling grou
p from bandits. And the happy story ends here. Have a nice day.
     2	Still here? okay, read on: The prince of Happalakkahuhu wished he could be
 as brave as his sister and vowed to train harder

$ # default folding width is 80
$ fold story.txt
The princess of a far away land fought bravely to rescue a travelling group from
 bandits. And the happy story ends here. Have a nice day.
Still here? okay, read on: The prince of Happalakkahuhu wished he could be as br
ave as his sister and vowed to train harder

$ fold story.txt | nl
     1	The princess of a far away land fought bravely to rescue a travelling grou
p from
     2	 bandits. And the happy story ends here. Have a nice day.
     3	Still here? okay, read on: The prince of Happalakkahuhu wished he could be
 as br
     4	ave as his sister and vowed to train harder
```

- `-s` option breaks at spaces to avoid word splitting

```
$ fold -s story.txt
The princess of a far away land fought bravely to rescue a travelling group
from bandits. And the happy story ends here. Have a nice day.
Still here? okay, read on: The prince of Happalakkahuhu wished he could be as
brave as his sister and vowed to train harder
```

- Use `-w` to change default width

```
$ fold -s -w60 story.txt
The princess of a far away land fought bravely to rescue a
travelling group from bandits. And the happy story ends
here. Have a nice day.
Still here? okay, read on: The prince of Happalakkahuhu
wished he could be as brave as his sister and vowed to
train harder
```

Further reading for fold

- `man fold` and `info fold` for more options and detailed documentation

File attributes

Table of Contents

wc

```
$ wc --version | head -n1
wc (GNU coreutils) 8.25

$ man wc
WC(1)                            User Commands                           WC(1)

NAME
       wc - print newline, word, and byte counts for each file

SYNOPSIS
       wc [OPTION]... [FILE]...
       wc [OPTION]... --files0-from=F

DESCRIPTION
       Print newline, word, and byte counts for each FILE, and a total line if
       more than one FILE is specified.  A word is a non-zero-length  sequence
       of characters delimited by white space.

       With no FILE, or when FILE is -, read standard input.
...
```

Various counts

```
$ cat sample.txt
Hello World
Good day
No doubt you like it too
Much ado about nothing
He he he

$ # by default, gives newline/word/byte count (in that order)
$ wc sample.txt
 5 17 78 sample.txt

$ # options to get individual numbers
$ wc -l sample.txt
5 sample.txt
$ wc -w sample.txt
17 sample.txt
$ wc -c sample.txt
78 sample.txt

$ # use shell input redirection if filename is not needed
$ wc -l < sample.txt
5
```

- multiple file input
- automatically displays total at end

```
$ cat greeting.txt
Hello there
Have a safe journey
$ cat fruits.txt
Fruit   Price
apple   42
banana  31
fig     90
guava   6

$ wc *.txt
  5  10  57 fruits.txt
  2   6  32 greeting.txt
  5  17  78 sample.txt
 12  33 167 total
```

- use `-L` to get length of longest line

```
$ wc -L < sample.txt
24

$ echo 'foo bar baz' | wc -L
11
$ echo 'hi there!' | wc -L
9

$ # last line will show max value, not sum of all input
$ wc -L *.txt
 13 fruits.txt
 19 greeting.txt
 24 sample.txt
 24 total
```

subtle differences

- byte count vs character count

```
$ # when input is ASCII
$ printf 'hi there' | wc -c
8
$ printf 'hi there' | wc -m
8

$ # when input has multi-byte characters
$ printf 'hi'  | od -x
0000000 6968 9ff0 8d91
0000006

$ printf 'hi'  | wc -m
3

$ printf 'hi'  | wc -c
6
```

- `-l` option gives only the count of number of newline characters

```
$ printf 'hi there\ngood day' | wc -l
1
$ printf 'hi there\ngood day\n' | wc -l
2
$ printf 'hi there\n\n\nfoo\n' | wc -l
4
```

- From `man wc` "A word is a non-zero-length sequence of characters delimited by white space"

```
$ echo 'foo        bar ;-*' | wc -w
3

$ # use other text processing as needed
$ echo 'foo        bar ;-*' | grep -iowE '[a-z]+'
foo
bar
$ echo 'foo        bar ;-*' | grep -iowE '[a-z]+' | wc -l
2
```

- `-L` won't count non-printable characters and tabs are converted to equivalent spaces

```
$ printf 'food\tgood' | wc -L
12
$ printf 'food\tgood' | wc -m
9
$ printf 'food\tgood' | awk '{print length()}'
9

$ printf 'foo\0bar\0baz' | wc -L
9
$ printf 'foo\0bar\0baz' | wc -m
11
$ printf 'foo\0bar\0baz' | awk '{print length()}'
11
```

Further reading for wc

- `man wc` and `info wc` for more options and detailed documentation
- wc Q&A on unix stackexchange
- wc Q&A on stackoverflow

du

```
$ du --version | head -n1
du (GNU coreutils) 8.25

$ man du
DU(1)                            User Commands                           DU(1)

NAME
       du - estimate file space usage

SYNOPSIS
       du [OPTION]... [FILE]...
       du [OPTION]... --files0-from=F

DESCRIPTION
       Summarize disk usage of the set of FILEs, recursively for directories.
...
```

Default size

- By default, size is given in size of **1024 bytes**
- Files are ignored, all directories and sub-directories are recursively reported

```
$ ls -F
projs/  py_learn@  words.txt

$ du
17920   ./projs/full_addr
14316   ./projs/half_addr
32952   ./projs
33880   .
```

- use `-a` to recursively show both files and directories
- use `-s` to show total directory size without descending into its sub-directories

```
$ du -a
712     ./projs/report.log
17916   ./projs/full_addr/faddr.v
17920   ./projs/full_addr
14312   ./projs/half_addr/haddr.v
14316   ./projs/half_addr
32952   ./projs
0       ./py_learn
924     ./words.txt
33880   .

$ du -s
33880   .

$ du -s projs words.txt
32952   projs
924     words.txt
```

- use `-S` to show directory size without taking into account size of its sub-directories

```
$ du -S
17920   ./projs/full_addr
14316   ./projs/half_addr
716     ./projs
928     .
```

Various size formats

```
$ # number of bytes
$ stat -c %s words.txt
938848
$ du -b words.txt
938848  words.txt

$ # kilobytes = 1024 bytes
$ du -sk projs
32952   projs
$ # megabytes = 1024 kilobytes
$ du -sm projs
33      projs

$ # -B to specify custom byte scale size
$ du -sB 5000 projs
6749    projs
$ du -sB 1048576 projs
33      projs
```

- human readable and si units

```
$ # in terms of powers of 1024
$ # M = 1048576 bytes and so on
$ du -sh projs/* words.txt
18M     projs/full_addr
14M     projs/half_addr
712K    projs/report.log
924K    words.txt

$ # in terms of powers of 1000
$ # M = 1000000 bytes and so on
$ du -s --si projs/* words.txt
19M     projs/full_addr
15M     projs/half_addr
730k    projs/report.log
947k    words.txt
```

- sorting

```
$ du -sh projs/* words.txt | sort -h
712K    projs/report.log
924K    words.txt
14M     projs/half_addr
18M     projs/full_addr

$ du -sk projs/* | sort -nr
17920   projs/full_addr
14316   projs/half_addr
712     projs/report.log
```

- to get size based on number of characters in file rather than disk space alloted

```
$ du -b words.txt
938848  words.txt

$ du -h words.txt
924K    words.txt

$ # 938848/1024 = 916.84
$ du --apparent-size -h words.txt
917K    words.txt
```

Dereferencing links

- See `man` and `info` pages for other related options

```
$ # -D to dereference command line argument
$ du py_learn
0       py_learn
$ du -shD py_learn
503M    py_learn

$ # -L to dereference links found by du
$ du -sh
34M     .
$ du -shL
536M    .
```

Filtering options

- `-d` to specify maximum depth

```
$ du -ah projs
712K    projs/report.log
18M     projs/full_addr/faddr.v
18M     projs/full_addr
14M     projs/half_addr/haddr.v
14M     projs/half_addr
33M     projs

$ du -ah -d1 projs
712K    projs/report.log
18M     projs/full_addr
14M     projs/half_addr
33M     projs
```

- `-c` to also show total size at end

```
$ du -cshD projs py_learn
33M     projs
503M    py_learn
535M    total
```

- `-t` to provide a threshold comparison

```
$ # >= 15M
$ du -Sh -t 15M
18M     ./projs/full_addr

$ # <= 1M
$ du -ah -t -1M
712K    ./projs/report.log
0       ./py_learn
924K    ./words.txt
```

- excluding files/directories based on **glob** pattern
- see also `--exclude-from=FILE` and `--files0-from=FILE` options

```
$ # note that excluded files affect directory size reported
$ du -ah --exclude='*addr*' projs
712K    projs/report.log
716K    projs

$ # depending on shell, brace expansion can be used
$ du -ah --exclude='*.'{v,log} projs
4.0K    projs/full_addr
4.0K    projs/half_addr
12K     projs
```

Further reading for du

- `man du` and `info du` for more options and detailed documentation
- du Q&A on unix stackexchange
- du Q&A on stackoverflow

df

```
$ df --version | head -n1
df (GNU coreutils) 8.25

$ man df
DF(1)                            User Commands                           DF(1)

NAME
       df - report file system disk space usage

SYNOPSIS
       df [OPTION]... [FILE]...

DESCRIPTION
       This  manual  page  documents  the  GNU version of df.  df displays the
       amount of disk space available on the file system containing each  file
       name  argument.   If  no file name is given, the space available on all
       currently mounted file systems is shown.
...
```

Examples

```
$ # use df without arguments to get information on all currently mounted file systems

$ df .
Filesystem     1K-blocks     Used Available Use% Mounted on
/dev/sda1       98298500 58563816  34734748  63% /

$ # use -B option for custom size
$ # use --si for size in powers of 1000 instead of 1024
$ df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1        94G   56G   34G  63% /
```

- Use `--output` to report only specific fields of interest

```
$ df -h --output=size,used,file / /media/learnbyexample/projs
 Size  Used File
  94G   56G /
  92G   35G /media/learnbyexample/projs

$ df -h --output=pcent .
Use%
 63%

$ df -h --output=pcent,fstype | awk -F'%' 'NR>2 && $1>=40'
 63% ext3
 40% ext4
 51% ext4
```

Further reading for df

- `man df` and `info df` for more options and detailed documentation
- df Q&A on stackoverflow
- Parsing df command output with awk
- processing df output

touch

```
$ touch --version | head -n1
touch (GNU coreutils) 8.25

$ man touch
TOUCH(1)                         User Commands                        TOUCH(1)

NAME
       touch - change file timestamps

SYNOPSIS
       touch [OPTION]... FILE...

DESCRIPTION
       Update  the  access  and modification times of each FILE to the current
       time.

       A FILE argument that does not exist is created empty, unless -c  or  -h
       is supplied.
...
```

Creating empty file

```
$ ls foo.txt
ls: cannot access 'foo.txt': No such file or directory
$ touch foo.txt
$ ls foo.txt
foo.txt

$ # use -c if new file shouldn't be created
$ rm foo.txt
$ touch -c foo.txt
$ ls foo.txt
ls: cannot access 'foo.txt': No such file or directory
```

Updating timestamps

- Updating both access and modification timestamp to current time

```
$ # last access time
$ stat -c %x fruits.txt
2017-07-19 17:06:01.523308599 +0530
$ # last modification time
$ stat -c %y fruits.txt
2017-07-13 13:54:03.576055933 +0530

$ touch fruits.txt
$ stat -c %x fruits.txt
2017-07-21 10:11:44.241921229 +0530
$ stat -c %y fruits.txt
2017-07-21 10:11:44.241921229 +0530
```

- Updating only access or modification timestamp

```
$ touch -a greeting.txt
$ stat -c %x greeting.txt
2017-07-21 10:14:08.457268564 +0530
$ stat -c %y greeting.txt
2017-07-13 13:54:26.004499660 +0530

$ touch -m sample.txt
$ stat -c %x sample.txt
2017-07-13 13:48:24.945450646 +0530
$ stat -c %y sample.txt
2017-07-21 10:14:40.770006144 +0530
```

- Using timestamp from another file to update

```
$ stat -c $'%x\n%y' power.log report.log
2017-07-19 10:48:03.978295434 +0530
2017-07-14 20:50:42.850887578 +0530
2017-06-24 13:00:31.773583923 +0530
2017-06-24 12:59:53.316751651 +0530

$ # copy both access and modification timestamp from power.log to report.log
$ touch -r power.log report.log
$ stat -c $'%x\n%y' report.log
2017-07-19 10:48:03.978295434 +0530
2017-07-14 20:50:42.850887578 +0530

$ # add -a or -m options to limit to only access or modification timestamp
```

- Using date string to update
- See also `-t` option

```
$ # add -a or -m as needed
$ touch -d '2010-03-17 17:04:23' report.log
$ stat -c $'%x\n%y' report.log
2010-03-17 17:04:23.000000000 +0530
2010-03-17 17:04:23.000000000 +0530
```

Preserving timestamp

- Text processing on files would update the timestamps

```
$ stat -c $'%x\n%y' power.log
2017-07-21 11:11:42.862874240 +0530
2017-07-13 21:31:53.496323704 +0530

$ sed -i 's/foo/bar/g' power.log
$ stat -c $'%x\n%y' power.log
2017-07-21 11:12:20.303504336 +0530
2017-07-21 11:12:20.303504336 +0530
```

- `touch` can be used to restore timestamps after processing

```
$ # first copy the timestamps using touch -r
$ stat -c $'%x\n%y' story.txt
2017-06-24 13:00:31.773583923 +0530
2017-06-24 12:59:53.316751651 +0530
$ # tmp.txt is temporary empty file
$ touch -r story.txt tmp.txt
$ stat -c $'%x\n%y' tmp.txt
2017-06-24 13:00:31.773583923 +0530
2017-06-24 12:59:53.316751651 +0530

$ # after text processing, copy back the timestamps and remove temporary file
$ sed -i 's/cat/dog/g' story.txt
$ touch -r tmp.txt story.txt && rm tmp.txt
$ stat -c $'%x\n%y' story.txt
2017-06-24 13:00:31.773583923 +0530
2017-06-24 12:59:53.316751651 +0530
```

Further reading for touch

- `man touch` and `info touch` for more options and detailed documentation

- touch Q&A on unix stackexchange

file

```
$ file --version | head -n1
file-5.25

$ man file
FILE(1)                   BSD General Commands Manual                  FILE(1)

NAME
     file — determine file type

SYNOPSIS
     file [-bcEhiklLNnprsvzZ0] [--apple] [--extension] [--mime-encoding]
          [--mime-type] [-e testname] [-F separator] [-f namefile]
          [-m magicfiles] [-P name=value] file ...
     file -C [-m magicfiles]
     file [--help]

DESCRIPTION
     This manual page documents version 5.25 of the file command.

     file tests each argument in an attempt to classify it.  There are three
     sets of tests, performed in this order: filesystem tests, magic tests,
     and language tests.  The first test that succeeds causes the file type to
     be printed.
...
```

File type examples

```
$ file sample.txt
sample.txt: ASCII text
$ # without file name in output
$ file -b sample.txt
ASCII text

$ printf 'hi\n'  | file -
/dev/stdin: UTF-8 Unicode text
$ printf 'hi\n'  | file -i -
/dev/stdin: text/plain; charset=utf-8

$ file ch
ch:  Bourne-Again shell script, ASCII text executable

$ file sunset.jpg moon.png
sunset.jpg: JPEG image data
moon.png: PNG image data, 32 x 32, 8-bit/color RGBA, non-interlaced
```

- different line terminators

```
$ printf 'hi' | file -
/dev/stdin: ASCII text, with no line terminators

$ printf 'hi\r' | file -
/dev/stdin: ASCII text, with CR line terminators

$ printf 'hi\r\n' | file -
/dev/stdin: ASCII text, with CRLF line terminators

$ printf 'hi\n' | file -
/dev/stdin: ASCII text
```

- find all files of particular type in current directory, for example `image` files

```
$ find -type f -exec bash -c '(file -b "$0" | grep -wq "image data") && echo "$0"' {
} \;
./sunset.jpg
./moon.png

$ # if filenames do not contain : or newline characters
$ find -type f -exec file {} + | awk -F: '/\<image data\>/{print $1}'
./sunset.jpg
./moon.png
```

Further reading for file

- `man file` and `info file` for more options and detailed documentation
- See also `identify` command which `describes the format and characteristics of one or more image files`

Miscellaneous

Table of Contents

cut

```
$ cut --version | head -n1
cut (GNU coreutils) 8.25

$ man cut
CUT(1)                           User Commands                          CUT(1)

NAME
       cut - remove sections from each line of files

SYNOPSIS
       cut OPTION... [FILE]...

DESCRIPTION
       Print selected parts of lines from each FILE to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

select specific fields

- Default delimiter is **tab** character
- `-f` option allows to print specific field(s) from each input line

```
$ printf 'foo\tbar\t123\tbaz\n'
foo     bar     123     baz

$ # single field
$ printf 'foo\tbar\t123\tbaz\n' | cut -f2
bar

$ # multiple fields can be specified by using ,
$ printf 'foo\tbar\t123\tbaz\n' | cut -f2,4
bar     baz

$ # output is always ascending order of field numbers
$ printf 'foo\tbar\t123\tbaz\n' | cut -f3,1
foo     123

$ # range can be specified using -
$ printf 'foo\tbar\t123\tbaz\n' | cut -f1-3
foo     bar     123
$ # if ending number is omitted, select till last field
$ printf 'foo\tbar\t123\tbaz\n' | cut -f3-
123     baz
```

suppressing lines without delimiter

```
$ cat marks.txt
jan 2017
foobar  12      45      23
feb 2017
foobar  18      38      19

$ # by default lines without delimiter will be printed
$ cut -f2- marks.txt
jan 2017
12      45      23
feb 2017
18      38      19

$ # use -s option to suppress such lines
$ cut -s -f2- marks.txt
12      45      23
18      38      19
```

specifying delimiters

- use `-d` option to specify input delimiter other than default **tab** character
- only single character can be used, for multi-character/regex based delimiter use `awk` or `perl`

```
$ echo 'foo:bar:123:baz' | cut -d: -f3
123

$ # by default output delimiter is same as input
$ echo 'foo:bar:123:baz' | cut -d: -f1,4
foo:baz

$ # quote the delimiter character if it clashes with shell special characters
$ 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
```

- use `--output-delimiter` option to specify different output delimiter
- since this option accepts a string, more than one character can be specified
- See also using $ prefixed string

```
$ printf 'foo\tbar\t123\tbaz\n' | cut --output-delimiter=: -f1-3
foo:bar:123

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

$ # tested on bash, might differ with other shells
$ echo 'one;two;three;four' | cut -d';' --output-delimiter=$'\t' -f1,3-
one     three   four

$ echo 'one;two;three;four' | cut -d';' --output-delimiter=' - ' -f1,3-
one - three - four
```

complement

```
$ echo 'one;two;three;four' | cut -d';' -f1,3-
one;three;four

$ # to print other than specified fields
$ echo 'one;two;three;four' | cut -d';' --complement -f2
one;three;four
```

select specific characters

- similar to `-f` for field selection, use `-c` for character selection
- See manual for what defines a character and differences between `-b` and `-c`

```
$ echo 'foo:bar:123:baz' | cut -c4
:

$ printf 'foo\tbar\t123\tbaz\n' | cut -c1,4,7
f       r

$ echo 'foo:bar:123:baz' | cut -c8-
:123:baz

$ echo 'foo:bar:123:baz' | cut --complement -c8-
foo:bar

$ echo 'foo:bar:123:baz' | cut -c1,6,7 --output-delimiter=' '
f a r

$ echo 'abcdefghij' | cut --output-delimiter='-' -c1-3,4-7,8-
abc-defg-hij

$ cut -c1-3 marks.txt
jan
foo
feb
foo
```

Further reading for cut

- `man cut` and `info cut` for more options and detailed documentation
- cut Q&A on unix stackexchange

tr

```
$ tr --version | head -n1
tr (GNU coreutils) 8.25

$ man tr
TR(1)                            User Commands                           TR(1)

NAME
       tr - translate or delete characters

SYNOPSIS
       tr [OPTION]... SET1 [SET2]

DESCRIPTION
       Translate, squeeze, and/or delete characters from standard input, writ-
       ing to standard output.
...
```

translation

- one-to-one mapping of characters, all occurrences are translated
- as good practice, enclose the arguments in single quotes to avoid issues due to shell interpretation

```
$ echo 'foo bar cat baz' | tr 'abc' '123'
foo 21r 31t 21z

$ # use - to represent a range in ascending order
$ echo 'foo bar cat baz' | tr 'a-f' '1-6'
6oo 21r 31t 21z

$ # changing case
$ echo 'foo bar cat baz' | tr 'a-z' 'A-Z'
FOO BAR CAT BAZ
$ echo 'Hello World' | tr 'a-zA-Z' 'A-Za-z'
hELLO wORLD

$ echo 'foo;bar;baz' | tr ; :
tr: missing operand
Try 'tr --help' for more information.
$ echo 'foo;bar;baz' | tr ';' ':'
foo:bar:baz
```

- rot13 example

```
$ echo 'foo bar cat baz' | tr 'a-z' 'n-za-m'
sbb one png onm
$ echo 'sbb one png onm' | tr 'a-z' 'n-za-m'
foo bar cat baz

$ 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
```

- use shell input redirection for file input

```
$ cat marks.txt
jan 2017
foobar  12      45      23
feb 2017
foobar  18      38      19

$ tr 'a-z' 'A-Z' < marks.txt
JAN 2017
FOOBAR  12      45      23
FEB 2017
FOOBAR  18      38      19
```

- if arguments are of different lengths

```
$ # when second argument is longer, the extra characters are ignored
$ echo 'foo bar cat baz' | tr 'abc' '1-9'
foo 21r 31t 21z

$ # when first argument is longer
$ # the last character of second argument gets re-used
$ echo 'foo bar cat baz' | tr 'a-z' '123'
333 213 313 213

$ # use -t option to truncate first argument to same length as second
$ echo 'foo bar cat baz' | tr -t 'a-z' '123'
foo 21r 31t 21z
```

escape sequences and character classes

- Certain characters like newline, tab, etc can be represented using escape sequences or octal representation
- Certain commonly useful groups of characters like alphabets, digits, punctuations etc have character class as shortcuts
- See gnu tr manual for all escape sequences and character classes

```
$ printf 'foo\tbar\t123\tbaz\n' | tr '\t' ':'
foo:bar:123:baz

$ echo 'foo:bar:123:baz' | tr ':' '\n'
foo
bar
123
baz
$ # makes it easier to transform
$ echo 'foo:bar:123:baz' | tr ':' '\n' | pr -2ats'-'
foo-bar
123-baz

$ echo 'foo bar cat baz' | tr '[:lower:]' '[:upper:]'
FOO BAR CAT BAZ
```

- since `-` is used for character ranges, place it at the end to represent it literally
  - cannot be used at start of argument as it would get treated as option
  - or use `--` to indicate end of option processing
- similarly, to represent `\` literally, use `\\`

```
$ echo '/foo-bar/baz/report' | tr '-a-z' '_A-Z'
tr: invalid option -- 'a'
Try 'tr --help' for more information.

$ echo '/foo-bar/baz/report' | tr 'a-z-' 'A-Z_'
/FOO_BAR/BAZ/REPORT

$ echo '/foo-bar/baz/report' | tr -- '-a-z' '_A-Z'
/FOO_BAR/BAZ/REPORT

$ echo '/foo-bar/baz/report' | tr '/-' '\\_'
\foo_bar\baz\report
```

deletion

- use `-d` option to specify characters to be deleted
- add complement option `-c` if it is easier to define which characters are to be retained

```
$ echo '2017-03-21' | tr -d '-'
20170321

$ echo 'Hi123 there. How a32re you' | tr -d '1-9'
Hi there. How are you

$ # delete all punctuation characters
$ echo '"Foo1!", "Bar.", ":Baz:"' | tr -d '[:punct:]'
Foo1 Bar Baz

$ # deleting carriage return character
$ cat -v greeting.txt
Hi there^M
How are you^M
$ tr -d '\r' < greeting.txt | cat -v
Hi there
How are you

$ # retain only alphabets, comma and newline characters
$ echo '"Foo1!", "Bar.", ":Baz:"' | tr -cd '[:alpha:],\n'
Foo,Bar,Baz
```

squeeze

- to change consecutive repeated characters to single copy of that character

```
$ # only lower case alphabets
$ echo 'FFoo seed 11233' | tr -s 'a-z'
FFo sed 11233

$ # alphabets and digits
$ echo 'FFoo seed 11233' | tr -s '[:alnum:]'
Fo sed 123

$ # squeeze other than alphabets
$ echo 'FFoo seed 11233' | tr -sc '[:alpha:]'
FFoo seed 123

$ # only characters present in second argument is used for squeeze
$ echo 'FFoo seed 11233' | tr -s 'A-Z' 'a-z'
fo sed 11233

$ # multiple consecutive horizontal spaces to single space
$ printf 'foo\t\tbar \t123     baz\n'
foo             bar     123     baz
$ printf 'foo\t\tbar \t123     baz\n' | tr -s '[:blank:]' ' '
foo bar 123 baz
```

Further reading for tr

- `man tr` and `info tr` for more options and detailed documentation
- tr Q&A on unix stackexchange

basename

```
$ basename --version | head -n1
basename (GNU coreutils) 8.25

$ man basename
BASENAME(1)                      User Commands                     BASENAME(1)

NAME
       basename - strip directory and suffix from filenames

SYNOPSIS
       basename NAME [SUFFIX]
       basename OPTION... NAME...

DESCRIPTION
       Print  NAME  with  any leading directory components removed.  If speci-
       fied, also remove a trailing SUFFIX.
...
```

Examples

```
$ # same as using pwd command
$ echo "$PWD"
/home/learnbyexample

$ basename "$PWD"
learnbyexample

$ # use -a option if there are multiple arguments
$ basename -a foo/a/report.log bar/y/power.log
report.log
power.log

$ # use single quotes if arguments contain space and other special shell characters
$ # use suffix option -s to strip file extension from filename
$ basename -s '.log' '/home/learnbyexample/proj adder/power.log'
power
$ # -a is implied when using -s option
$ basename -s'.log' foo/a/report.log bar/y/power.log
report
power
```

- Can also use Parameter expansion if working on file paths saved in variables
  - assumes `bash` shell and similar that support this feature

```
$ # remove from start of string up to last /
$ file='/home/learnbyexample/proj adder/power.log'
$ basename "$file"
power.log
$ echo "${file##*/}"
power.log

$ t="${file##*/}"
$ # remove .log from end of string
$ echo "${t%.log}"
power
```

- See `man basename` and `info basename` for detailed documentation

dirname

```
$ dirname --version | head -n1
dirname (GNU coreutils) 8.25

$ man dirname
DIRNAME(1)                       User Commands                      DIRNAME(1)

NAME
       dirname - strip last component from file name

SYNOPSIS
       dirname [OPTION] NAME...

DESCRIPTION
       Output each NAME with its last non-slash component and trailing slashes
       removed; if NAME contains no  /'s,  output  '.'  (meaning  the  current
       directory).
...
```

Examples

```
$ echo "$PWD"
/home/learnbyexample

$ dirname "$PWD"
/home

$ # use single quotes if arguments contain space and other special shell characters
$ dirname '/home/learnbyexample/proj adder/power.log'
/home/learnbyexample/proj adder

$ # unlike basename, by default dirname handles multiple arguments
$ dirname foo/a/report.log bar/y/power.log
foo/a
bar/y

$ # if no / in argument, output is . to indicate current directory
$ dirname power.log
.
```

- Use `$()` command substitution to further process output as needed

```
$ dirname '/home/learnbyexample/proj adder/power.log'
/home/learnbyexample/proj adder

$ dirname "$(dirname '/home/learnbyexample/proj adder/power.log')"
/home/learnbyexample

$ basename "$(dirname '/home/learnbyexample/proj adder/power.log')"
proj adder
```

- Can also use Parameter expansion if working on file paths saved in variables
  - assumes `bash` shell and similar that support this feature

```
$ # remove from last / in the string to end of string
$ file='/home/learnbyexample/proj adder/power.log'
$ dirname "$file"
/home/learnbyexample/proj adder
$ echo "${file%/*}"
/home/learnbyexample/proj adder

$ # remove from second last / to end of string
$ echo "${file%/*/*}"
/home/learnbyexample

$ # apply basename trick to get just directory name instead of full path
$ t="${file%/*}"
$ echo "${t##*/}"
proj adder
```

- See `man dirname` and `info dirname` for detailed documentation

xargs

```
$ xargs --version | head -n1
xargs (GNU findutils) 4.7.0-git

$ whatis xargs
xargs (1)            - build and execute command lines from standard input

$ # from 'man xargs'
       This manual page documents the GNU version of xargs.  xargs reads items
       from  the  standard  input, delimited by blanks (which can be protected
       with double or single quotes or a backslash) or newlines, and  executes
       the  command (default is /bin/echo) one or more times with any initial-
       arguments followed by items read from standard input.  Blank  lines  on
       the standard input are ignored.
```

While `xargs` is primarily used for passing output of command or file contents to another command as input arguments and/or parallel processing, it can be quite handy for certain text processing stuff with default `echo` command

```
$ printf ' foo\t\tbar \t123     baz \n' | cat -e
 foo        bar     123     baz $
$ # tr helps to change consecutive blanks to single space
$ # but what if blanks at start and end have to be removed as well?
$ printf ' foo\t\tbar \t123     baz \n' | tr -s '[:blank:]' ' ' | cat -e
 foo bar 123 baz $
$ # xargs does this by default
$ printf ' foo\t\tbar \t123     baz \n' | xargs | cat -e
foo bar 123 baz$

$ # -n option limits number of arguments per line
$ printf ' foo\t\tbar \t123     baz \n' | xargs -n2
foo bar
123 baz

$ # same as using: paste -d' ' - - -
$ # or: pr -3ats' '
$ seq 6 | xargs -n3
1 2 3
4 5 6
```

- use `-a` option to specify file input instead of stdin

```
$ cat marks.txt
jan 2017
foobar  12      45      23
feb 2017
foobar  18      38      19

$ xargs -a marks.txt
jan 2017 foobar 12 45 23 feb 2017 foobar 18 38 19

$ # use -L option to limit max number of lines per command line
$ xargs -L2 -a marks.txt
jan 2017 foobar 12 45 23
feb 2017 foobar 18 38 19
```

- **Note** since `echo` is the command being executed, it will cause issue with option interpretation

```
$ printf ' -e foo\t\tbar \t123     baz \n' | xargs -n2
foo
bar 123
baz

$ # use -t option to see what is happening (verbose output)
$ printf ' -e foo\t\tbar \t123     baz \n' | xargs -n2 -t
echo -e foo
foo
echo bar 123
bar 123
echo baz
baz
```

- See `man xargs` and `info xargs` for detailed documentation

seq

```
$ seq --version | head -n1
seq (GNU coreutils) 8.25

$ man seq
SEQ(1)                           User Commands                          SEQ(1)

NAME
       seq - print a sequence of numbers

SYNOPSIS
       seq [OPTION]... LAST
       seq [OPTION]... FIRST LAST
       seq [OPTION]... FIRST INCREMENT LAST

DESCRIPTION
       Print numbers from FIRST to LAST, in steps of INCREMENT.
...
```

integer sequences

- see `info seq` for details of how large numbers are handled
    - for ex: `seq 50000000000000000000 2 50000000000000000004` may not work

```
$ # default start=1 and increment=1
$ seq 3
1
2
3

$ # default increment=1
$ seq 25434 25437
25434
25435
25436
25437
$ seq -5 -3
-5
-4
-3

$ # different increment value
$ seq 1000 5 1011
1000
1005
1010

$ # use negative increment for descending order
$ seq 10 -5 -7
10
5
0
-5
```

- use `-w` option for leading zeros
- largest length of start/end value is used to determine padding

```
$ seq 008 010
8
9
10

$ # or: seq -w 8 010
$ seq -w 008 010
008
009
010

$ seq -w 0003
0001
0002
0003
```

specifying separator

- As seen already, default is newline separator between numbers
- `-s` option allows to use custom string between numbers
- A newline is always added at end

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

$ seq -s' ' 4
1 2 3 4

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

floating point sequences

```
$ # default increment=1
$ seq 0.5 2.5
0.5
1.5
2.5

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

$ # Scientific notation is supported
$ seq 1.2e2 1.22e2
120
121
122
```

- formatting numbers, see `info seq` for details

```
$ seq -f'%.3f' -s':' -2 0.75 3
-2.000:-1.250:-0.500:0.250:1.000:1.750:2.500

$ seq -f'%.3e' 1.2e2 1.22e2
1.200e+02
1.210e+02
1.220e+02
```

Further reading for seq

- `man seq` and `info seq` for more options, corner cases and detailed documentation
- seq Q&A on unix stackexchange