Filters and file operations

When working with (text) files we often want to perform operations to manipulate the file contents to get a specific output.

For example: We have a huge logfile that logs all login attemps for our application where it logs usernames, ip adresses, timestamps, location, ISP info, metadata, ... We might want to quickly lookup all incorrect login attemps by user X and see what IP addresses he was connecting from. We could manually go through the file line by line to check it but often that process is long and tedious. What we want to do is use commands to filter, manipulate and structure the data to the desired result. To do this we can use a wide set of filter and structure commands that we can link together using *pipes*.

For the examples in this chapter we will use a log file filled with data. To download this file you can run the following command:

bash

```
student@linux-ess:~$ wget https://d-ries.github.io/linux-essentials/data/
--2022-06-19 18:15:24--  https://d-ries.github.io/linux-essentials/data/a
Resolving d-ries.github.io (d-ries.github.io)... 185.199.109.153, 185.199
Connecting to d-ries.github.io (d-ries.github.io)|185.199.109.153|:443...
HTTP request sent, awaiting response... 200 OK
Length: 1730 (1.7K) [text/plain]
Saving to: 'auth.log'

auth.log                      100%[=====================================

2022-06-19 18:15:25 (5.32 MB/s) - 'auth.log' saved [1730/1730]
student@linux-ess:~$ cat auth.log
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09 11:11:11 linux-ess: Server listening on :: port 22.
Jun 09 12:32:24 linux-ess: Accepted publickey for: johndoe from 85.245.10
Jun 10 21:38:01 linux-ess: Failed password for: student from 192.168.0.1
Jun 10 21:39:01 linux-ess: Failed password for: johndoe from 192.168.0.2
```

```
Jun 10 21:42:01 linux-ess: Accepted password for: student from 84.298.138
Jun 14 14:12:33 linux-ess: Accepted password for: johndoe from 192.168.0.
Jun 14 14:14:12 linux-ess: Accepted publickey for: student from 85.245.10
Jun 15 17:42:18 linux-ess: Failed password for: janedoe from 192.168.0.10
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
Jun 19 22:43:23 linux-ess: Failed password for: johndoe from 85.245.107.4
Jun 22 08:04:00 linux-ess: Accepted password for: johndoe from 192.168.0.
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

Using pipes

A pipe (`|`) is a specific symbol that we can use to link commands together. The pipe symbol will take the `stdout` from the previous command and fowards it to the `stdin` of the next command:

bash

```
student@linux-ess:~$ head -3 auth.log | tail -2
Jun 09 11:11:11 linux-ess: Server listening on :: port 22.
Jun 09 12:32:24 linux-ess: Accepted publickey for: johndoe from 85.245.10
```

The example above will run the `head -3` command which will take the first 3 lines of the file `auth.log` . The output containing the first 3 lines of the file will then be used as input for the `tail -2` command which results in taking the bottom 2 lines of the first 3 lines of the file `auth.log` . This means that the result is the second and third line of the file.

You can use as many pipes as you want in a command line. It will just keep passing the output of a command to the input of the next command and so on:

```bash
student@linux-ess:~$ cat auth.log | head | tail -3 | head -2 | tail -1
Jun 15 17:42:18 linux-ess: Failed password for: janedoe from 192.168.0.10
```

Write to file (tee)

Sometimes you want to temporarily write the results to a seperate file while continuing to work with pipes to get a certain end result. This command will forward the `stdin` to the `stdout` but will also store the `stdin` in a file provided as an argument:

```bash
student@linux-ess:~$ tail -3 auth.log | tee temp_log | tail -1
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ cat temp_log
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

tee is also often used to put results in a file where you don't have the right privileges and need to use `sudo` :

bash

```
student@linux-ess:~$ tail -3 auth.log | head -1 > /filteredlogfile
-bash: /filteredlogfile: Permission denied
student@linux-ess:~$ sudo tail -3 auth.log | head -1 > /filteredlogfile
-bash: /filteredlogfile: Permission denied
student@linux-ess:~$ tail -3 auth.log | head -1 | tee /filteredlogfile
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
tee: /filteredlogfile: Permission denied
student@linux-ess:~$ cat /filteredlogfile
cat: /filteredlogfile: No such file or directory
student@linux-ess:~$ tail -3 auth.log | head -1 | sudo tee /filteredlogfi
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
student@linux-ess:~$ cat /filteredlogfile
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
```

If you want to add to the output file instead of overwriting it you can specify the option `-a`

bash

```
student@linux-ess:~$ head -1 auth.log | sudo tee -a /filteredlogfile
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
student@linux-ess:~$ cat /filteredlogfile
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
```

Filtering output

Using content filters (grep)

We often want to browse the contents of a file and retaining the lines containing a certain string or pattern. This is where the `grep` command comes in. `grep` is one of the most used filter commands in Linux systems. We can use it as a standalone command as follows:

bash

```
student@linux-ess:~$ grep jane auth.log
Jun 15 17:42:18 linux-ess: Failed password for: janedoe from 192.168.0.10
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
```

Or we can use it as a *pipe*:

bash

```
student@linux-ess:~$ cat auth.log | grep jane
Jun 15 17:42:18 linux-ess: Failed password for: janedoe from 192.168.0.10
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
```

Both commands give the same result and work the same. What we can see is that the `grep` command will filter the file contents based on a string or pattern (in this case the string `jane`).

> An important note to make is that, by default, `grep` is a case sensitive command. It will only show the lines in the file containing that specific keyword. Note that searching for the string *failed* in all lower case will result in 0 lines being returned:

bash

```
student@linux-ess:~$ cat auth.log | grep failed
student@linux-ess:~$
```

However there is an option `-i` to make grep work case insensitive:

bash

```
student@linux-ess:~$ cat auth.log | grep -i failed
Jun 10 21:38:01 linux-ess: Failed password for: student from 192.168.0.1
Jun 10 21:39:01 linux-ess: Failed password for: johndoe from 192.168.0.2
Jun 15 17:42:18 linux-ess: Failed password for: janedoe from 192.168.0.10
Jun 19 22:43:23 linux-ess: Failed password for: johndoe from 85.245.107.4
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
```

```
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
```

Another interesting option is `-v` which will return all lines *not* containing the string/pattern:

bash

```
student@linux-ess:~$ cat auth.log | grep -i -v failed
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09 11:11:11 linux-ess: Server listening on :: port 22.
Jun 09 12:32:24 linux-ess: Accepted publickey for: johndoe from 85.245.10
Jun 10 21:42:01 linux-ess: Accepted password for: student from 84.298.138
Jun 14 14:12:33 linux-ess: Accepted password for: johndoe from 192.168.0.
Jun 14 14:14:12 linux-ess: Accepted publickey for: student from 85.245.10
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
Jun 22 08:04:00 linux-ess: Accepted password for: johndoe from 192.168.0.
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

Knowing that we can use multiple pipes (`|`) in a command, we can combine multiple grep commands as well. Imagine the scenario where we want to find only successful login attemps made by the user `janedoe` . We could do this as follows:

bash

```
student@linux-ess:~$ cat auth.log | grep -vi failed | grep janedoe
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
```

or (there are multiple correct answers here)

bash

```
student@linux-ess:~$ cat auth.log | grep janedoe | grep Accepted
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
```

Sometimes we might want to see more than just the line that contains the string/pattern. Imagine we want to see some more context as to where the file is located. We can customize the grep command to show lines `before` and/or `after` the result as follows:

bash

```
student@linux-ess:~$ cat example.log
this is line one
this is line two
this is line three
this is line four
this is line five
student@linux-ess:~$ cat example.log | grep -A1 three
this is line three
this is line four
student@linux-ess:~$ cat example.log  | grep -B2 four
this is line two
this is line three
this is line four
student@linux-ess:~$ cat example.log | grep -C1 three
this is line two
this is line three
this is line four
```

Another important note to make is that filters only work on the output stream and not on the error stream. So if we want to search through the errors as well we have to combine the two streams:

bash

```
student@linux-ess:~$ find /etc/ssl -name "*.c??"
/etc/ssl/openssl.cnf
/etc/ssl/certs/ca-certificates.crt
find: ‘/etc/ssl/private’: Permission denied
student@linux-ess:~$ find /etc/ssl -name "*.c??" | tr 'abcde' 'ABCDE'
/EtC/ssl/opEnssl.Cnf
/EtC/ssl/CErts/CA-CErtifiCAtEs.Crt
find: ‘/etc/ssl/private’: Permission denied
```

```
student@linux-ess:~$ find /etc/ssl -name "*.c??" |& tr 'abcde' 'ABCDE'
/EtC/ssl/opEnssl.Cnf
/EtC/ssl/CErts/CA-CErtifiCAtEs.Crt
finD: ‘/EtC/ssl/privAtE’: PErmission DEniED
```

Or another example where we only want to see the debug files from the sys folder that have *kernel* in the name. We see here that it prints all the error lines because the grep filter does not work on the error stream

bash

```
student@linux-ess:~$ find /sys -iname "*kernel*" | grep debug
find: '/sys/kernel/tracing': Permission denied
find: '/sys/kernel/debug': Permission denied
find: '/sys/fs/pstore': Permission denied
find: '/sys/fs/bpf': Permission denied
/sys/fs/cgroup/sys-kernel-debug.mount
```

The solution is again to merge the two streams together:

bash

```
student@linux-ess:~$ find /sys -iname "*kernel*" |& grep debug
find: '/sys/kernel/debug': Permission denied
/sys/fs/cgroup/sys-kernel-debug.mount
```

Regular expressions

In the examples above we only used simple strings to find certain lines in a file. Sometimes we want to filter on dynamic content. Imagine finding all logins from an ip address containing '192' followed by other characters, or finding users that have "doe" as a lastname. In these case we will search for strings via a certain pattern. To achieve this we have to use a dynamic syntax called a regular expression.

Regular expressions can turn into a real rabbit hole. We will only focus on the most used cases and a couple of practical examples but know that there is a

whole *regex* world to be explored that is beyond the scope of this course!

The `grep` command can use different kinds of *regex* patterns. By default it uses *basic regular expressions (BRE)* but for a lot of cases we want to extend this to *extended regular expressions (ERE)*. To do this we have to use the `-E` option in the `grep` command. This gives us a lot more functionality when it comes to building dynamic search queries. A lot of the commands explained below won't work without the `-E` option!

For the examples used in this (sub)chapter we will use a seperate file that you can download using the command below:

```bash
student@linux-ess:~$ wget https://d-ries.github.io/linux-essentials/data/
--2022-11-03 19:49:29--  https://d-ries.github.io/linux-essentials/data/r
Resolving d-ries.github.io (d-ries.github.io)... 185.199.110.153, 185.199
Connecting to d-ries.github.io (d-ries.github.io)|185.199.110.153|:443...
HTTP request sent, awaiting response... 200 OK
Length: 317 [text/plain]
Saving to: ‘regexlist.txt’

regexlist.txt                               100%[========================

2022-11-03 19:49:29 (588 KB/s) - ‘regexlist.txt’ saved [317/317]
student@linux-ess:~$ cat regexlist.txt
Charlotte
Lawrence
Max
Stan
Emma
Sara
John
Kelly
Ian
Ellen
Tim
Robin
Nora
Tom
```

```
Caroline
Michael
john.doe@pxl.b
john.doe@pxl.be
p
pl
px
pxl
pxxl
pxxxl
pxxxxl
128
32
64
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
192.168.1.19
192.168.5.117
172.16.0.4
127.0.0.1
pxe
pxe boot
This is a test.
This has been tested
```

To start of we will use some special symbols that we've used before. We've seen the impact of an asterisk (`*`) in the chapter about *file globbing*. An asterisk has a similar functionality in a regex but there are some important key differences:

- In file globbing a `*` sign means 0, one or more of any type of character
- In a regex, a `*` sign means 0, one or more of the previous character

Take the example below. We expect only the `pxl` variants to show up, but as we can see in the output all of the file contents show. This is because every line matches the regex `zero, one or more of the letter p` :

```
student@linux-ess:~$ cat regexlist.txt | grep "p*"
Charlotte
Lawrence
Max
Stan
Emma
Sara
John
Kelly
Ian
Ellen
Tim
Robin
Nora
Tom
Caroline
Michael
john.doe@pxl.b
john.doe@pxl.be
p
pl
px
pxl
pxxl
pxxxl
pxxxxl
128
32
64
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
192.168.1.19
192.168.5.117
172.16.0.4
127.0.0.1
pxe
pxe boot
```

```
This is a test.
This has been tested
```

If we want to filter the lines with a `p` and the following character might be an `x` this is done by using the following syntax:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "px*"
john.doe@pxl.b
john.doe@pxl.be
p
pl
px
pxl
pxxl
pxxxl
pxxxxl
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
pxe
pxe boot
```

Notice that the line doesn't have to start with the pattern.

Because of the fact that we do not put any characters behind the x we could also not specify this character:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "p"
john.doe@pxl.b
john.doe@pxl.be
p
pl
px
pxl
pxxl
```

```
pxxxl
pxxxxl
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
pxe
pxe boot
```

Now we tell the regex to find lines that contain a `p` followed by zero, one or more `x` characters. This is exactly why `pl` shows up (it contains zero of the character x):

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "px*l"
john.doe@pxl.b
john.doe@pxl.be
pl
pxl
pxxl
pxxxl
pxxxxl
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
```

Imagine if we wanted to use a regex that contains one or more of a character rather than zero, one or more. We can do this using the `+` sign. If we use this we will see that the line with the text `pl` isn't in the results anymore:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -E "px+l"
john.doe@pxl.b
john.doe@pxl.be
pxl
pxxl
pxxxl
pxxxxl
```

```
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
```

Note that if we don't use the -E option here we have to escape the plus sign (+). ... | grep "px\+l"

To take it even a step further, what about exactly 3 occurences? Easy, we can do this as follows:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -E "px{3}l"
pxxxl
```

The `{3}` is linked to the character before that.

Note that if we don't use the -E option here we have to escape the curly braces ({}). ... | grep "px\{3\}l"

Imagine now we wanted to check for lines that start or end with a specific character or character set. We'll start of with lines starting with a specific character:

bash

```
student@linux-ess:~$  cat regexlist.txt | grep -i "^[smc]"
Charlotte
Max
Stan
Sara
Caroline
Michael
```

The example above uses a `^` sign that indicates the start of a line. Next up we

use square brackets `[ ]` that we can use to specify characters that can be used as the start of the line. In this case the letters `S`, `M`, and `C`. We could also use ranges:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "^[0-9]"
128
32
64
192.168.1.19
192.168.5.117
172.16.0.4
127.0.0.1
student@linux-ess:~$ cat regexlist.txt | grep "^[a-z]"
john.doe@pxl.b
john.doe@pxl.be
p
pl
px
pxl
pxxl
pxxxl
pxxxxl
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
pxe
pxe boot
```

> The square brackets are not linked to the beginning and end of a line, so you can use them wherever in the regex. Be aware that these are case sensitive!

To check for lines ending with a specific character we can use a `$` sign:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "e$"
```

```
Charlotte
Lawrence
Caroline
john.doe@pxl.be
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
pxe
student@linux-ess:~$ cat regexlist.txt | grep "[0-9]$"
128
32
64
192.168.1.19
192.168.5.117
172.16.0.4
127.0.0.1
```

So what about matching exactly 1 character? A `.` translates to exactly one character of any type:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -i "t.m"
Tim
Tom
```

We can combine this with starts & endings as well:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "^...\."
192.168.1.19
192.168.5.117
172.16.0.4
127.0.0.1
```

This translates to **`start with any type of character`** (`^.`) followed by 2 more characters of any type (`..`), followed by a regular dot (`\.`). Notice how

we escaped the last dot so it doesn't get interpreted as a special regex character!

If we want to filter the lines that have one or another pattern we could use the character `|` :

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -E "^M|n$"
Max
Stan
John
Ian
Ellen
Robin
Michael
```

Here we search for lines beginning with an `M` or ending with an `n`

We could do the same with using the option -E multiple times:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -e "^M" -e "n$"
Max
Stan
John
Ian
Ellen
Robin
Michael
```

If we want to filter lines that comply with multiple patterns we could use the command grep multiple times:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep "^E" | grep "a$"
Emma
```

Here we search for lines ending with an `e` and beginning with a `C`

If we want to filter lines that only have the searchstring as a whole word we use the option `-w` :

```bash
student@ubuntu-server:~$ cat regexlist.txt | grep -w "test"
This is a test.
```

Here we search for lines with `test` as a single word

Pattern examples

Creating a regex that checks for a IPv4 address:

```bash
student@linux-ess:~$ cat regexlist.txt | grep -E "^[0-9]{1,3}\.[0-9]{1,3}
192.168.1.19
192.168.5.117
172.16.0.4
127.0.0.1
```

Note that this does not validate a valid IPv4 address, whether or not it is public or private or if the number ranges are in the `1-255` range.

> The `{1,3}` syntax means that the previous characters need to appear between 1 and 3 times!

> The `^` at the beginning and `$` at the end means that nothing else is allowed on the same line!

Creating a regex that checks for a valid e-mail address format:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -E "^.+\@[a-zA-Z0-9]+\.[a-z
john.doe@pxl.be
```

Creating a regex that checks for a valid url format:

bash

```
student@linux-ess:~$ cat regexlist.txt | grep -E "https?://.+\."
http://www.pxl.be
https://www.pxl.be
```

Note that this example does not check for valid domain names.

> The questionmark (`?`) in a regex means the previous character is *optional*!

Using content structure (cut,sort,uniq)

Using columns (cut)

Using the `cut` command we can split lines in a file into columns. To do this we have to define a delimiter (this is a character that defines the start of a new column) for example a `space` or `:` sign. Then we can select which columns the command should display:

bash

```
student@linux-ess:~$ cat auth.log
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09 11:11:11 linux-ess: Server listening on :: port 22.
Jun 09 12:32:24 linux-ess: Accepted publickey for: johndoe from 85.245.10
Jun 10 21:38:01 linux-ess: Failed password for: student from 192.168.0.1
Jun 10 21:39:01 linux-ess: Failed password for: johndoe from 192.168.0.2
Jun 10 21:42:01 linux-ess: Accepted password for: student from 84.298.138
Jun 14 14:12:33 linux-ess: Accepted password for: johndoe from 192.168.0.
Jun 14 14:14:12 linux-ess: Accepted publickey for: student from 85.245.10
```

```
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
Jun 19 22:43:23 linux-ess: Failed password for: johndoe from 85.245.107.4
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ head -3 auth.log | cut -d":" -f1
Jun 09 11
Jun 09 11
Jun 09 12
```

As you can see in the example above we used the `:` sign as the delimiter. We then used the `-f` option to only display the first column. A full line in this log file looks like this:

```
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
```

This means that the first column ends on the `:` sign in the time notation (after the hour). The second column exists out of the minutes part of the time notation, the third column the seconds of the time notation and the hostname and so on. We can also tell the command to display multiple columns:

bash

```
student@linux-ess:~$ head -3 auth.log | cut -d":" -f1,2,3
Jun 09 11:11:11 linux-ess
Jun 09 11:11:11 linux-ess
Jun 09 12:32:24 linux-ess
```

Another nice example of where we can use this is the `/etc/passwd` file where we can easily filter all the usernames and there homefolder locations:

bash

```
student@linux-ess:~$ cat /etc/passwd | grep bash$
root:x:0:0:root:/root:/bin/bash
student:x:1000:1000:student:/home/student:/bin/bash
student@linux-ess:~$ cat /etc/passwd | grep bash$  | cut -d":" -f1,6
root:/root
student:/home/student
```

Note that the homefolder of the user root isn't in the folder /home, but within the root of the filesystem (/)

Using sorting (sort & uniq)

If we want to sort the lines of a file we can use the `sort` command. With the `uniq` command only the unique values will be shown.

bash

```
student@linux-ess:~$ cat auth.log | grep -E "Accepted|Failed"
Jun 09 12:32:24 linux-ess: Accepted publickey for: johndoe from 85.245.10
Jun 10 21:38:01 linux-ess: Failed password for: student from 192.168.0.1
Jun 10 21:39:01 linux-ess: Failed password for: johndoe from 192.168.0.2
Jun 10 21:42:01 linux-ess: Accepted password for: student from 84.298.138
Jun 14 14:12:33 linux-ess: Accepted password for: johndoe from 192.168.0.
Jun 14 14:14:12 linux-ess: Accepted publickey for: student from 85.245.10
Jun 17 18:22:22 linux-ess: Accepted password for: janedoe from 192.168.0.
Jun 19 22:43:23 linux-ess: Failed password for: johndoe from 85.245.107.4
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ cat auth.log | grep -E "Accepted|Failed" | cut -d ':
doeg
janedoe
```

```
johndoe
student
```

Note that to use `uniq` you must allways first `sort` the data!

Note that the pipe sign (|) can be used as the `OR` operator. If we want to apply the `AND` operator we can pipe two greps after each other or use a regex:

```bash
cat auth.log | grep "port 44293"
Jun 22 21:11:11 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ cat auth.log | grep -i "accepted" | grep "port 44293
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ cat auth.log | grep -i "accepted.*port 44293"
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

Using counts (wc)

If we want to count the lines, words or bytes of a document we can use the command `wc` :

```bash
student@linux-ess:~$ wc auth.log
  17  245 1731 auth.log
```

The first number is the number of lines, the second number the number of words and the third number the number of bytes of the file content.

We can also output only one of those:

bash

```
student@linux-ess:~$ wc -l auth.log # numer of lines
17 auth.log
student@linux-ess:~$ wc -w auth.log # number of words
245 auth.log
student@linux-ess:~$ wc -c auth.log # number of bytes
1731 auth.log
```

Manipulating output

Translate (tr)

The `tr` command allows us to *translate* certain characters to other characters. It takes in 2 arguments:

- The character (or set of characters) that needs to be replaced
- The character (or set of characters) the previous set needs to be replaced by

We can see a simple example below:

bash

```
student@linux-ess:~$ head -3 auth.log | tr 'e' 'E'
Jun 09 11:11:11 linux-Ess: SErvEr listEning on 0.0.0.0 port 22.
Jun 09 11:11:11 linux-Ess: SErvEr listEning on :: port 22.
Jun 09 12:32:24 linux-Ess: AccEptEd publickEy for: johndoE from 85.245.10
```

all of the letters `e` in the output will be translated to the capital `E`. As said in the explanation above we can use ranges or sets of characters as well:

bash

```
student@linux-ess:~$ head -3 auth.log | tr 'abcdklmn' 'ABCDKLMN'
JuN 09 11:11:11 LiNux-ess: Server ListeNiNg oN 0.0.0.0 port 22.
JuN 09 11:11:11 LiNux-ess: Server ListeNiNg oN :: port 22.
JuN 09 12:32:24 LiNux-ess: ACCepteD puBLiCKey for: johNDoe froM 85.245.10
student@linux-ess:~$ head -3 auth.log | tr 'a-z' 'A-Z'
```

```
JUN 09 11:11:11 LINUX-ESS: SERVER LISTENING ON 0.0.0.0 PORT 22.
JUN 09 11:11:11 LINUX-ESS: SERVER LISTENING ON :: PORT 22.
JUN 09 12:32:24 LINUX-ESS: ACCEPTED PUBLICKEY FOR: JOHNDOE FROM 85.245.10
```

It's not just regular characters we can replace. We can use special characters such as newlines or tabs as follows:

bash

```
student@linux-ess:~$ head -3 auth.log | od -c
0000000   J   u   n       0   9       1   1   :   1   1   :   1   1
0000020   l   i   n   u   x   -   e   s   s   :       S   e   r   v   e
0000040   r       l   i   s   t   e   n   i   n   g       o   n       0
0000060   .   0   .   0   .   0       p   o   r   t       2   2   .  \n
0000100   J   u   n       0   9       1   1   :   1   1   :   1   1
0000120   l   i   n   u   x   -   e   s   s   :       S   e   r   v   e
0000140   r       l   i   s   t   e   n   i   n   g       o   n       :
0000160   :       p   o   r   t       2   2   .  \n   J   u   n       0
0000200   9       1   2   :   3   2   :   2   4       l   i   n   u   x
0000220   -   e   s   s   :       A   c   c   e   p   t   e   d       p
0000240   u   b   l   i   c   k   e   y       f   o   r   :       j   o
0000260   h   n   d   o   e       f   r   o   m       8   5   .   2   4
0000300   5   .   1   0   7   .   4   2       p   o   r   t       5   4
0000320   2   5   9       s   s   h   2   :       R   S   A       S   H
0000340   A   2   5   6   :   K   1   8   k   P   G   Z   r   T   i   z
0000360   7   g   >  \n
0000364
student@linux-ess:~$ head -3 auth.log | tr '\n' ' '
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22. Jun 09 11
```

The example above changes all the *newlines* to *spaces*.

Imagine we have some data that has multiple occurances of a specific character and we only want it to display one (=squeeze). We can use the `-s` option to delete any recurring characters:

bash

```
student@linux-ess:~$ head -3 auth.log | tr -s '1'
Jun 09 1:1:1 linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09 1:1:1 linux-ess: Server listening on :: port 22.
Jun 09 12:32:24 linux-ess: Accepted publickey for: johndoe from 85.245.10
```

lastly we have the option `-d` that will *delete* any of the character(s) we define in the text:

bash

```
student@linux-ess:~$ head -3 auth.log | tr -d ':'
Jun 09 111111 linux-ess Server listening on 0.0.0.0 port 22.
Jun 09 111111 linux-ess Server listening on  port 22.
Jun 09 123224 linux-ess Accepted publickey for johndoe from 85.245.107.42
```

Stream editor (sed)

`sed` is an advanced stream editor that can perform editing functions using regular expressions. Remember the `rename` command? We can use the same syntax of regular expressions here:

bash

```
student@linux-ess:~$ tail -4 auth.log | sed 's/Failed/Incorrect/'
Jun 22 21:11:12 linux-ess: Incorrect password for: doeg from 192.168.0.10
Jun 22 21:11:12 linux-ess: Incorrect password for: doeg from 192.168.0.10
Jun 22 21:11:12 linux-ess: Incorrect password for: doeg from 192.168.0.10
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

The example above will change the string `Failed` to `Incorrect` once for every line.

> Note that only in the output the changes are applied. The file itself isn't altered. If you want to apply it on the file itself you can add the option `-i` :

bash

```
student@linux-ess:~$ tail -4 auth.log > lastfourlines
student@linux-ess:~$ cat lastfourlines
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 pc
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ sed -i 's/Failed/Incorrect/' lastfourlines
student@linux-ess:~$ cat lastfourlines
Jun 22 21:11:12 linux-ess: Incorrect password for: doeg from 192.168.0.10
Jun 22 21:11:12 linux-ess: Incorrect password for: doeg from 192.168.0.10
Jun 22 21:11:12 linux-ess: Incorrect password for: doeg from 192.168.0.10
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

Mind that by default sed will only change the string once per line. This is something we need to be aware of as we can see in the example below:

bash

```
student@linux-ess:~$ echo "example this is an example" | sed 's/example/t
test this is an example
```

Only the first occurence of `example` is changed to `test`. If we want the `sed` command to change every occurence in a line, we have to use the `g` (global) flag as seen below:

bash

```
student@linux-ess:~$ echo "example this is an example" | sed 's/example/t
test this is an test
```

There is also an `i` flag that will make the regex (=search string) case insensitive:

bash

```
student@linux-ess:~$ echo "example this is an Example" | sed 's/example/t
```

```
test this is an Example
student@linux-ess:~$ echo "example this is an Example" | sed 's/example/t
test this is an test
```

We're also able to use sed to mask certain text:

bash

```
student@ubuntu-server:~$ head -2 auth.log
Jun 09 11:11:11 linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09 11:11:11 linux-ess: Server listening on :: port 22.
student@ubuntu-server:~$ head -2 auth.log | sed 's/..:..:../00:00:00/'
Jun 09 00:00:00 linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09 00:00:00 linux-ess: Server listening on :: port 22.
student@ubuntu-server:~$ head -2 auth.log | sed 's/..:..:..//'
Jun 09  linux-ess: Server listening on 0.0.0.0 port 22.
Jun 09  linux-ess: Server listening on :: port 22.
```

Lastly we will look at the `d` flag which will remove any line containing the string:

bash

```
student@linux-ess:~$ tail -4 auth.log
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Failed password for: doeg from 192.168.0.10 po
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
student@linux-ess:~$ tail -4 auth.log | sed '/Failed/d'
Jun 22 21:11:12 linux-ess: Accepted password for: doeg from 192.168.0.10
```

Offcourse we can use our knowledge of regular expressions with sed, but if you want to use extended regular expressions you need to specify the option `-r` :

bash

```
student@ubuntu-server:~$ grep -C2 www regexlist.txt
32
64
```

```
htp://www.pxl.be
http://www.pxl.be
https://www.pxl.be
192.168.1.19
192.168.5.117
student@ubuntu-server:~$ grep -C2 www regexlist.txt | sed -r 's_https?://
32
64
htp://www.pxl.be
url masked
url masked
192.168.1.19
192.168.5.117
```

Because we use slashes (/) in our regex we can opt to use underscores (_) as seperator

Previous
7 Advanced command structures

Next
Lab