Navigating Linux System Commands

A guide for beginners to the Shell and GNU coreutils

Sayan Ghosh

July 20, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on July 20, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

List of Figures

List of Tables

1 Grep

We have already seen and used `grep` throughout this chapter while discussing regex. `grep` is a command that is used to search for patterns in a file. It is used to search for a pattern in a file, and print the lines that match the pattern.

The name `grep` comes from the **g/re/p** command in the **ed** editor. The **ed** editor is a line editor, and the **g/re/p** command is used to search for a pattern in a file, and print the lines that match the pattern. The `grep` command is a standalone command that is used to search for a pattern in a file, and print the lines that match the pattern.

`grep` can use **BRE** or **ERE**, or even **PCRE** if the `-P` flag is used. By default, `grep` matches using the **BRE** engine.

```
$ grep "[aeiou]" -o <<< "hello how are you?"
e
o
o
a
e
o
u
```


1.1 Regex Engine

However, `grep` can also use the **ERE** engine using the `-E` flag. This is useful when we want to use the special characters like `+`, `?`, `(`, `)`, `{`, `}`, and `|` without escaping them. [1]

```
$ grep -E "p+" -o <<< "apple and pineapples"
pp
p
pp
```

Sometimes, it is required to not use regex at all, and simply search for a string. We can use the `-F` flag to search for a fixed string, and not a regex. This will search for any line that has the substring. Any symbol that has special meaning in regex will be treated as a literal character.

```
$ grep -F "a+b*c=?" -o <<< "What is a+b*c=?"
a+b*c=?
```

This mode is useful if you want to match any arbritary string in a file, and do not know what the string is going to be, thus not allowing you to escape the special characters.

```
$ cat data.txt
Hello, this is a file
with a lot of equations
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
3. F = ma
```

1: There is also an executable called `egrep`, which is the same as `grep -E`. It is provided for compatibility with older Unix systems. It is not recommended to use `egrep`, as it is deprecated and not present in all systems. You should use `grep -E` instead.

```
4. The meaning of life, the universe, and everything = 42
$ read -r -p "What to search for? " pattern
What to search for? ^2
$ grep "$pattern" data.txt
2. E = mc^2
$ grep -F "$pattern" data.txt
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
```

In the above example, you can see a file full of equations, thus special symbols. If we want to dynamically input what to search for, we can use the `read` command to read the input from the user, and then use `grep` to search for the pattern. If we use the `-F` flag, we can search for the pattern as a fixed string, and not as a regex.

Here we wanted to find all the quadratic equations, and thus searched for `^2`. However, observe that the `grep`, without the `-F` flag, did not match the first equation which has multiple `^2` in it, because it is treating `^` with its special meaning of **start of line anchor**.

If we were statically providing the search string, we can simply escape the special characters, and use `grep` without the `-F` flag. But in cases like this, it is easier to simply use the `-F` to not use regular expression and simply find substrings.

1.2 PCRE

Similarly, there are situations where the ERE engine is not powerful enough and we need to use the **PCRE** engine. We can use the `-P` flag to use the PCRE engine.

```
$ cat data.txt
Hello, this is a file
with a lot of equations
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
3. F = ma
4. The meaning of life, the universe, and everything = 42
$ grep -P "c\^2" data.txt
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
$ grep -P "c\^2(?=.*triangle)" data.txt
1. a^2 + b^2 = c^2 for right angle triangles
```

Here, if we want to find all the equations with `c^2`, but only if that equation also mentions "triangle" somewhere after the `c^2`, then we can use lookahead assertions of PCRE to accomplish that. Here, the `.*` matches any character zero or more times, and the `triangle` matches the string "triangle". Putting it inside a lookahead assertion (`(?= )`) ensures that the pattern is present, but not part of the match.

This can be confirmed by using the `-o` flag to print only the matched part of the line.

```
$ grep -P "c\^2(?=.*triangle)" -o data.txt
c^2
```

1.3 Print Only Matching Part

We have been using the `-o` flag extensively to print only the matching part of the line. This is useful when we want to extract only the part of the line that matches the pattern and not the entire line. This is also very useful for debugging regex, as we can see what part of the line is actually matching the pattern.

If we do not use `-o`, then any line having one or more match will be printed entirely, and the matches will be colored red [2] ,however, if two consequtive matches are present, it becomes hard to distinguish between the two matches.

If we use the `-o` flag, then only the matching part of the line is printed, and each match is printed on a new line, making it easy to see exactly which parts are matched.

```
$ grep -E "o{,3}" <<< "hellooooo"
hellooooo
$ grep -Eo "o{,3}" <<< "hellooooo"
ooo
oo
```

In the above example, we ask `grep` to match the pattern `o{,3}`, that is, match the letter `o` zero to three times. If we do not use the `-o` flag, then the entire line is printed, and the matches are colored red. This however creates a confusion, even though all the 5 *o*'s are matched, they cannot obviously be a single match, since a single match can only be of a maximum of 3 *o*'s. So how are the matches grouped? Is it 3 *o*'s followed by 1 *o* followed by another *o*? Is it 2 *o*'s followed by 2 *o*'s followed by one *o*? There seems to be no way to tell from the first output.

However, if we use the `-o` flag, then only the matching part of the line is printed, and each match is printed on a new line. It then becomes clear that grep will greedily match as much as possible first, so the first three *o*'s are matched, and then the remaining are grouped as a single match.

2: `grep` will color the match only if you pass the flag `--color=always`, or if the flag is set to `--color=auto` and the terminal supports color output. Otherwise no ANSI-escape code is printed by grep.

You can also change the color of the match by setting the `GREP_COLORS` environment variable. The default color is red, but you can change it to any color you want.

1.4 Matching Multiple Patterns

1.4.1 Disjunction

If we want to match multiple patterns, we can use the `-e` flag to specify multiple patterns. This is useful when we want to match multiple patterns, and not just a single pattern. Any line containing one or more of the patterns we are searching for will be printed. This is like using an **OR** clause.

```
$ cat data.txt
Hello, this is a file
with a lot of equations
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
3. F = ma
4. The meaning of life, the universe, and everything = 42
$ grep "file" data.txt
```

In this example, we want to find lines that match the word "file" or the word "life". We can use the `-e` flag to specify multiple patterns. The `-e` flag is automatically implied if we are searching for a single pattern, however, if we are searching for more than one pattern, we have specify it for all of the patterns, including the first one.

```
Hello, this is a file
$ grep -e "file" -e "life" data.txt
Hello, this is a file
4. The meaning of life, the universe, and everything = 42
```

1.4.2 Conjunction

If we want to match lines that contain all of the patterns we are searching for, we can pipe the output of one `grep` to another, to do iterative filtering.

If we want to find lines which start with a number, and also contain the word "a", then we can use two greps to accomplish this.

In this example, first we show each of the pattern we are matching, and that they output multiple lines. Then finally we combine both the patterns using pipe to find only the common line in both the outputs. The patterns can be specified in either order. We need to provide the file only in the first grep, and we should **not** provide a file name to any of the other greps, otherwise it will not use the output of the previous grep as input and just filter from the file instead.

Here the regex `^[0-9]` matches any line that starts with a number. And the regex `\ba\b` matches any line that contains the word "a". The `\b` is a word boundary, and matches the start or end of a word. Thus it will match the word "a" and not the letter "a" in a word.

```
$ cat data.txt
Hello, this is a file
with a lot of equations
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
3. F = ma
4. The meaning of life, the universe, and everything = 42
$ grep '\ba\b' data.txt
Hello, this is a file
with a lot of equations
1. a^2 + b^2 = c^2 for right angle triangles
$ grep '^[0-9]' data.txt
1. a^2 + b^2 = c^2 for right angle triangles
2. E = mc^2
3. F = ma
4. The meaning of life, the universe, and everything = 42
$ grep '\ba\b' data.txt  | grep '[0-9]'
1. a^2 + b^2 = c^2 for right angle triangles
```

1.5 Read Patterns from File

If we have a lot of patterns to search for, we can put them in a file, and use the `-f` flag to read the patterns from the file. Each line of the file is treated as a separate pattern, and any line that matches any of the patterns will be printed. The type of regex engine used depends on whether `-E`, `-P`, or `-F` is used.

```
$ cat data.txt
p+q=r
apple
e*f=g
$ cat pattern
p+
e*
$ grep -G -f pattern data.txt  -o
p+
e
e
$ grep -F -f pattern data.txt  -o
p+
e*
```

```
$ grep -E -f pattern data.txt  -o
p
pp
e
e
```

1.6 Ignore Case

If we want to ignore the case of the pattern, we can use the `-i` flag. This is useful when we want to match a pattern, but do not care about the case of the pattern, or we are not sure what the case is.

```
$ grep 'apple' /usr/share/dict/words | head
appleberry
appleblossom
applecart
apple-cheeked
appled
appledrane
appledrone
apple-eating
apple-faced
apple-fallow
$ grep -i 'apple' /usr/share/dict/words | head
Apple
appleberry
Appleby
appleblossom
applecart
apple-cheeked
appled
Appledorf
appledrane
appledrone
```

As seen above, the first `grep` matches only the lines that contain the word "apple", and not "Apple". However, the second `grep` matches both "apple" and "Apple".

1.7 Invert Match

Sometimes its easier to specify the patterns we do not want to match, rather than the patterns we want to match. We can use the `-v` flag to invert the match, that is, to print only the lines that do not match the pattern.

```
$ cat data.txt
apple
banana
blueberry
blackberry
raspberry
strawberry
$ grep -v 'berry' data.txt
apple
```

```
banana
```

This is useful when we want to filter out some arbritary stream of data for some patterns.

In this example, we are filtering out all the users that have the shell set to `nologin`. These are users which cannot be logged into. We can use the `-v` flag to invert the match, and print only the users that do not have the shell set to `nologin`. Effectively printing all the accounts in the current system that can be logged into. **ntp** uses the `/bin/false` shell, which is used to prevent the user from logging in, but is not the same as `/usr/bin/nologin`, which is used to prevent the user from logging in and also prints a message to the user.

```
$ grep -v 'nologin$' /etc/passwd
root:x:0:0:root:/root:/usr/bin/bash
git:x:971:971:git daemon user:/:/usr/bin/git-shell
ntp:x:87:87:Network Time Protocol:/var/lib/ntp:/bin/false
sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
test1:x:1001:1002::/home/test1:/usr/bin/bash
```

1.8 Anchoring

If we want to match a pattern only at the start of the line, or at the end of the line, we can use the `^` and `$` anchors respectively.

However, in `grep`, if we want to match a pattern that is the entire line, we can use the `-x` flag. This is useful when we want to match the entire line, and not just a part of the line. This is same as wrapping the entire pattern in `^$`, but is more readable.

Similarly, if we want to match a pattern that is a word, we can use the `-w` flag. This is useful when we want to match a word, and not a part of a word. This is same as wrapping the entire pattern in `\\b`, but is more readable.

Observe in this example, if we do not use the `-w` flag then words that have the substring "apple" will also be matched. However, when we use the `-w` flag, only the word "apple" is matched as a whole word.

```
$ grep 'apple' /usr/share/dict/words | tail
stapple
star-apple
strapple
thorn-apple
thrapple
toffee-apple
undappled
ungrapple
ungrappled
ungrappler
$ grep -w 'apple' /usr/share/dict/words | tail
may-apple
oak-apple
pine-apple
pond-apple
rose-apple
snap-apple
sorb-apple
star-apple
thorn-apple
toffee-apple
```

1.9 Counting Matches

Sometimes all we want is to see how many lines match the pattern, and not the lines themselves. We can use the `-c` flag to count the number of

lines that match the pattern. This is exactly same as piping the output of `grep` to `wc -l`, but is more readable.

```
$ grep -c 'apple' /usr/share/dict/words
101
$ grep -ic 'apple' /usr/share/dict/words
107
```

From this we can quickly see that there are $107 - 101 = 6$ lines that contain the word "Apple" in the file.

We can also print those lines using the `diff` command.

```
$ diff <(grep apple /usr/share/dict/words) <(grep -i apple /usr/
    share/dict/words)
0a1
> Apple
1a3
> Appleby
5a8
> Appledorf
10a14
> Applegate
28a33
> Appleseed
31a37
> Appleton
```

Or by using the `comm` command.

The `comm` command is used to compare two sorted files line by line. It is used to find lines that are common, or different between two files. The `comm` command requires the input files to be **sorted**, and it will not work if the files are not sorted. The `comm` command has three numbered columns, the first column is the lines that are unique to the first file, the second column is the lines that are unique to the second file, and the third column is the lines that are common to both files. The `comm` command is useful when we want to compare two files, and find the differences between them, or find the lines common between them.

```
$ comm -13 <(grep apple /usr/share/dict/words | sort) <(grep -i
    apple /usr/share/dict/words | sort)
Apple
Appleby
Appledorf
Applegate
Appleseed
Appleton
```

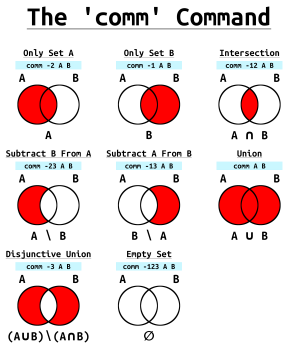

Figure 1.1: The `comm` command

Observe that we had to sort the files before using `comm`, as `comm` requires the files to be sorted. We can use the `<(command)` syntax to pass the output of a command as a file to another command. This is called process substitution.

1.10 Print Filename

Sometimes, we may want to search for a pattern in multiple files, and we want to know which file contains the pattern. We can use the `-H` flag to print the filename along with the matched line. This is however the default behaviour in GNU grep if multiple files are passed.

```
$ cat hello.txt
hello world
```

In this example, we are passing all the `.txt` files in the current directory to `grep`, and searching for patterns. The `-H` flag is implicit and is used to print the filename along with the matched line. This is useful when we are searching for a pattern in multiple files, and we want to know which file contains the pattern.

```
hello universe
$ cat linux.txt
this is linux
$ grep hello *txt
hello.txt:hello world
hello.txt:hello universe
$ grep this *txt
linux.txt:this is linux
$ grep i *txt
hello.txt:hello universe
linux.txt:this is linux
```

But if we want to suppress the printing of the filename, we can use the `-h` flag. This is useful when we are searching for a pattern in multiple files, and we do not want to know which file contains the pattern, just the line.

```
$ cat hello.txt
hello world
hello universe
$ cat linux.txt
this is linux
$ grep -h hello *txt
hello world
hello universe
$ grep -h this *txt
this is linux
$ grep -h i *txt
hello universe
this is linux
```

Similarly, if we want to print the filename only if there are multiple files, we can use the `-l` flag. This will print **only** the name of the file that has one or more matches. This mode does not print the filename multiple time even if it has multiple matches on same or different lines. Thus this is not same as `grep pattern files... | cut -d: -f1`. Rather it is same as `grep pattern files... | cut -d: -f1 | uniq`

1.11 Limiting Output

Although we can use `head` and `tail` to limit the number of lines of output of `grep`, `grep` also has the `-m` flag to limit the number of matches. This is useful when we want to see only the first few matches, and not all the matches.

The benefit of using `-m` instead of `head` is that the output will remain colored if coloring is supported, although it is not visible here, try running both to observe the difference.

```
$ grep 'nologin' /etc/passwd
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
nobody:x:65534:65534:Kernel Overflow User:/:/usr/bin/nologin
dbus:x:81:81:System Message Bus:/:/usr/bin/nologin
systemd-coredump:x:984:984:systemd Core Dumper:/:/usr/bin/nologin
systemd-network:x:982:982:systemd Network Management:/:/usr/bin/
    nologin
```

```
systemd-oom:x:981:981:systemd Userspace OOM Killer:/:/usr/bin/
    nologin
systemd-journal-remote:x:980:980:systemd Journal Remote:/:/usr/bin
    /nologin
systemd-journal-upload:x:979:979:systemd Journal Upload:/:/usr/bin
    /nologin
systemd-resolve:x:978:978:systemd Resolver:/:/usr/bin/nologin
systemd-timesync:x:977:977:systemd Time Synchronization:/:/usr/bin
    /nologin
tss:x:976:976:tss user for tpm2:/:/usr/bin/nologin
uuidd:x:68:68::/:/usr/bin/nologin
avahi:x:974:974:Avahi mDNS/DNS-SD daemon:/:/usr/bin/nologin
named:x:40:40:BIND DNS Server:/:/usr/bin/nologin
dnsmasq:x:973:973:dnsmasq daemon:/:/usr/bin/nologin
geoclue:x:972:972:Geoinformation service:/var/lib/geoclue:/usr/bin
    /nologin
_talkd:x:970:970:User for legacy talkd server:/:/usr/bin/nologin
nbd:x:969:969:Network Block Device:/var/empty:/usr/bin/nologin
nm-openconnect:x:968:968:NetworkManager OpenConnect:/:/usr/bin/
    nologin
nm-openvpn:x:967:967:NetworkManager OpenVPN:/:/usr/bin/nologin
nvidia-persistenced:x:143:143:NVIDIA Persistence Daemon:/:/usr/bin
    /nologin
openvpn:x:965:965:OpenVPN:/:/usr/bin/nologin
partimag:x:110:110:Partimage user:/:/usr/bin/nologin
polkitd:x:102:102:PolicyKit daemon:/:/usr/bin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/usr/bin/nologin
rpcuser:x:34:34:RPC Service User:/var/lib/nfs:/usr/bin/nologin
rtkit:x:133:133:RealtimeKit:/proc:/usr/bin/nologin
sddm:x:964:964:SDDM Greeter Account:/var/lib/sddm:/usr/bin/nologin
usbmux:x:140:140:usbmux user:/:/usr/bin/nologin
qemu:x:962:962:QEMU user:/:/usr/bin/nologin
cups:x:209:209:cups helper user:/:/usr/bin/nologin
dhcpcd:x:959:959:dhcpcd privilege separation:/:/usr/bin/nologin
redis:x:958:958:Redis in-memory data structure store:/var/lib/
    redis:/usr/bin/nologin
saned:x:957:957:SANE daemon user:/:/usr/bin/nologin
tor:x:43:43::/var/lib/tor:/usr/bin/nologin
$ grep 'nologin' /etc/passwd | head -n5
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
$ grep 'nologin' /etc/passwd -m5
bin:x:1:1::/:/usr/bin/nologin
daemon:x:2:2::/:/usr/bin/nologin
mail:x:8:12::/var/spool/mail:/usr/bin/nologin
ftp:x:14:11::/srv/ftp:/usr/bin/nologin
http:x:33:33::/srv/http:/usr/bin/nologin
```

```
$ cat hello.txt
hello world
hello universe
$ cat linux.txt
this is linux
$ grep -l hello *txt
hello.txt
$ grep -l this *txt
```

```
linux.txt
$ grep -l i *txt
hello.txt
linux.txt
```

1.12 Quiet Quitting

No, we are not talking about the recent trend of doing the bare minimum in a job.

If we want to suppress the output of `grep`, and only see if the pattern was found or not, we can use the `-q` flag. This is useful when we want to use `grep` in a script, and we do not want to see the output of `grep`, just the exit status. This also implies `-m1`, as we can determine the exit status of `grep` as soon as we find the first match.

In this example we use concepts such as `if` and `then` to check the exit status of `grep`. If the exit status is 0, then the pattern was found, and we print "world mentioned". If the exit status is 1, then the pattern was not found, and we do not print anything.
`$?` is a special variable that stores the exit status of the last command. If the exit status is 0, then the command was successful, and if the exit status is 1, then the command failed.
We will cover these in later chapters.

```
$ cat hello.txt
hello world
hello universe
$ grep -q 'world' hello.txt
$ echo $?
0
$ if grep -q 'world' hello.txt ; then echo "world mentioned" ; fi
world mentioned
$ if grep -q 'galaxy' hello.txt ; then echo "galaxy mentioned" ;
    fi
```

1.13 Numbering Lines

If we want to number the lines that match the pattern, we can use the `-n` flag. This is useful when we want to see the line number of the lines that match the pattern.

```
$ cat hello.txt
hello world
hello universe
$ grep -n 'hello' hello.txt
1:hello world
2:hello universe
```

1.14 Recursive Search

`grep` can also search for patterns in directories and subdirectories recursively. We can use the `-r` flag to achieve this. This is useful when we want to search for a pattern in multiple files, and we do not know which files contain the pattern, or if the files are nested deeply inside directories.

By default it starts searching from the current directory, but we can specify the directory to start searching from as the second argument.

```
$ mkdir -p path/to/some/deeply/nested/folders/like/this
mkdir: created directory 'path'
mkdir: created directory 'path/to'
mkdir: created directory 'path/to/some'
mkdir: created directory 'path/to/some/deeply'
mkdir: created directory 'path/to/some/deeply/nested'
mkdir: created directory 'path/to/some/deeply/nested/folders'
mkdir: created directory 'path/to/some/deeply/nested/folders/like'
mkdir: created directory 'path/to/some/deeply/nested/folders/like/
    this'
$ echo hello > path/to/some/deeply/nested/folders/like/this/hello.
    txt
$ echo "hello world" > path/world.txt
$ grep -r hello
path/world.txt:hello world
path/to/some/deeply/nested/folders/like/this/hello.txt:hello
$ grep -r hello path/to/
path/to/some/deeply/nested/folders/like/this/hello.txt:hello
```

1.15 Context Line Control

Sometimes it is useful to see a few lines before or after the actual line that contains the matched pattern. We can use the `-A`, `-B`, and `-C` flags to control the number of lines to print after, before, and around the matched line respectively. [3]

3: The `-A` flag is for printing lines **after** the match, the `-B` flag is for printing lines **before** the match, and the `-C` flag is for printing lines both **before** and **after** the match.

```
$ grep -n sayan /etc/passwd
37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
$ grep -n sayan /etc/passwd -A2
37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
38-qemu:x:962:962:QEMU user:/:/usr/bin/nologin
39-cups:x:209:209:cups helper user:/:/usr/bin/nologin
$ grep -n sayan /etc/passwd -B2
35-sddm:x:964:964:SDDM Greeter Account:/var/lib/sddm:/usr/bin/
    nologin
36-usbmux:x:140:140:usbmux user:/:/usr/bin/nologin
37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
$ grep -n sayan /etc/passwd -C2
35-sddm:x:964:964:SDDM Greeter Account:/var/lib/sddm:/usr/bin/
    nologin
36-usbmux:x:140:140:usbmux user:/:/usr/bin/nologin
37:sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
38-qemu:x:962:962:QEMU user:/:/usr/bin/nologin
39-cups:x:209:209:cups helper user:/:/usr/bin/nologin
```

These were some of the flags that can be used with `grep` to control the output. There are many more flags that can be used with `grep`, and you can see them by running `man grep`.

Now that we have learnt some of the flags on their own, let us try to accomplish certain tasks using multiple flags together.

1.16 Finding Lines Common in Two Files

If we have two files which has some lines of text, and our job is to find the lines that are common in both, we can use `comm` if the files are sorted, or we can sort the files before passing it to `comm` if we are allowed to have the output sorted.

But if we are not allowed to sort the files, we can use `grep` to accomplish this.

```
$ cat file1.txt
this is file1
this.*
a common line
is to check if it misinteprets regex
apple
$ cat file2.txt
this is file2
this.*
a common line
is to check if we are checking fixed strings
pineapple
```

Ideally the lines that should be detected as being common are the lines

```
this.*
a common line
```

Remember that grep allows using a file as the pattern, and it will match any line that matches any of the patterns in the file. Let us try to use it to provide one file as a pattern, and the other file as the input.

```
$ grep -f file2.txt file1.txt
this is file1
this.*
a common line
```

Observe that it is also matching the line `this is file1`, which is not common in both the files. This is because the `.*` in the pattern `this.*` is being interpreted as a regex, and not as a literal character. We can use the `-F` flag to treat the pattern as a fixed string, and not as a regex.

```
$ grep -Ff file2.txt file1.txt
this.*
a common line
```

So is that it? It looks like we are getting the correct output. Not yet. If we reverse the order of files, we see that we are not getting the correct output.

```
$ grep -Ff file1.txt file2.txt
this.*
a common line
pineapple
```

This is because the `apple` in `file1.txt` is matching the word `pineapple` in `file2.txt` as a substring.

If we want to print only lines that match the entire line, we can use the `-x` flag.

```
$  grep -Fxf file1.txt file2.txt
this.*
a common line
```

Now we are getting the correct output, in the original file order.

If we were to use `comm`, we would have to sort the files first, and then use `comm` to find the common lines.

```
$ comm -12 file1.txt  file2.txt
comm: file 1 is not in sorted order
comm: file 2 is not in sorted order
comm: input is not in sorted order
$ comm -12 <( sort file1.txt)  <(sort file2.txt)
a common line
this.*
```

Observe how the order of output is different.

The trick to mastering grep is to understand the flags, and how they can be combined to accomplish the task at hand. The more you practice, the more you will understand how to use grep effectively. Refer to the practice questions in the VM.