

Linux Commands

Goals for today's workshop are :

1. Overcome *fear of the command-line*.
2. Learn usable skills that you will use every day and that you can honestly and confidently add to your resume.
3. Continue your forward progression with the momentum that got you here in the first place.

Some General Characteristics of Linux:

- 1) Linux is **case-sensitive**, unlike Windows... all commands in Linux are *lower case*.
- 2) *Almost all* Linux commands are abbreviations or acronyms. A few commands are the actual word.
- 3) Linux is a "*no-news-is-good-news*" kind of system; if a command succeeds, you get the expected output, if any, followed by another command prompt. In other words, there are *no success messages*. If an error occurs then Linux displays the appropriate error message.
- 4) Almost all Linux commands have **options** that can change the way the command operates (*we'll see this as we go, below*). Options are invoked by following the command with a hyphen and the a letter and/or number.
- 5) Linux is very **extensible**, meaning that as you start working with more apps or services there are CLIs (command-line interfaces) that you can install, and most of these CLIs make use of what you already know.
- 6) Experience with Linux command and skills tend to build on themselves, and you may find yourself having many 'Ah-ha!' moments as you learn more (*some folks find that actually exciting and quite satisfying*).

As we go through this session we will be revisiting some commands and applying new skills or command options.

Linux Commands

In this session we'll learn Linux **Commands** and **Skills** that help you feel confident and proficient at the command-line.

Commands:

For example :

`ls -l` (long format list)

`ls -latr` (long format list, all files, sort by time, reverse order)

Skills:

Skills are often **keyboard shortcuts**, but can also sometimes be **other commands** and especially **habits**, which can be combined to make working on the command line way less scary, much more safe, and even kind of fun!

1) Habits:

- a) When first starting a command-line session, use the `ls -latr` command to see where you are in the file system and also to get your mindset into command-line mode
- b) Always make a back-up of a file you're about to change with the `cp` command.
- c) When using wildcards to impact a group of files, verify the wildcard search string first using the `ls` command to see which files would be affected.
- d) Understanding the File structure (hierarchy) and how to navigate it easily

2) Keyboard Shortcuts

- a) Up / Down cursor arrow key to recall previous commands
- b) Tab key (code completion)

3) Commands

- a) `history` (command)
- b) Understanding the output of the `ls -l` (long format) command
- c) Using the `tree` command to understand the directory structure

4) Other

- a) Output redirection to a file using the ">" and ">>"
- b) Output redirection from one command to the input of another command using the '|' pipe symbol

Linux Commands

Learning Plan for This Session

Sections:

- 1) `ls`, `touch`, `cat`, `less` (list-storage, touch, concatenate)
- 2) `cp`, `mv`, `rm` (copy, move | rename, remove i.e., delete)
- 3) `pwd`, `tree`, `cd`, `mkdir`, `rmdir` (print-working-directory, make directory, remove-directory change-directory, view file structure)
- 4) `history`, `grep`, `man`, `--help` (command history, global regular expression print, manual, quick help)
- 5) Redirection of output to a file using either `>` or `>>`
 - a) from screen to a file
 - b) from `echo` or `print` command to a file
 - c) The `|` (pipe symbol) → passes output from one command into the input of another command.
Mixing & matching redirect and pipe
- 6) Working with the `history` command buffer
 - a) `!` (single exclamation) → Works in conjunction with the `history` command to execute a specific command number
 - b) `!!` (double-exclamation) → recall previous command
 - c) `wc` (word count) → returns the count of words (...or lines or characters, depending on the option used) in a file or the output from another command.

Linux Commands

SECTION 1:

ls and touch

1. **ls** (*list storage*): Lists the files in the current directory (*or whichever directory you specify*)
2. **touch**: This command actually does two things.

First, it can create new, zero-byte (*empty*) files.

We actually need to create some files to play with in this lab so we'll be learning and using the touch command:

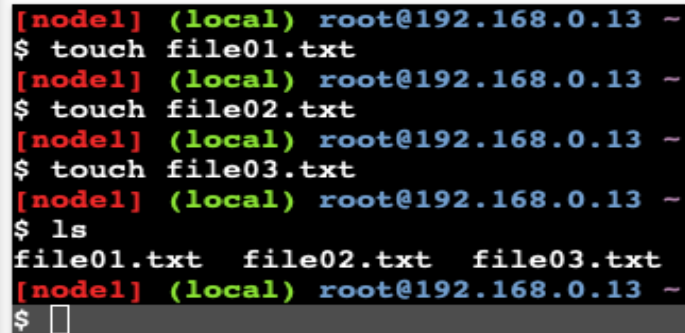
Example:

touch file01.txt

touch file02.txt

touch file03.txt

When we execute the **ls** command we'll see the files we just created:



```
[node1] (local) root@192.168.0.13 ~
$ touch file01.txt
[node1] (local) root@192.168.0.13 ~
$ touch file02.txt
[node1] (local) root@192.168.0.13 ~
$ touch file03.txt
[node1] (local) root@192.168.0.13 ~
$ ls
file01.txt  file02.txt  file03.txt
[node1] (local) root@192.168.0.13 ~
$
```

Second, the **touch** command can also update the date/time stamp of an existing or several files without changing the contents of the file.

This second function is used in system administration functions like batch processing jobs, but it's still important to know that this capability exists. For example, a job can inspect the date/time stamp of a file to see when the last time it was modified and then either run additional steps (e.g., it's time to run some processes) or to ignore it since it's not yet time.

Now that we've introduced these commands, let's *play* with them (I'm a firm believer in playing vs. studying. I think you can learn more when you're actively having fun while just experimenting than trying to follow along some step-by-step lesson).

Feel free to try or ask about things as we go.

Linux Commands

SKILL: Use the Up-arrow key and Down-arrow key to recall previous commands and traverse up and down the command stack (*more on this below ... history command*).

The **ls** command displays a list of file names only and it does so from left to right, not line by line as we might expect.

To see more details about each file, use **ls -l** (*long format*) option.

```
$ ls
file01.txt file02.txt file03.txt
[rodel] (local) root@192.168.0.13
ls -l
total 0
-rw-r--r-- 1 root root 0 Dec 21 14:29 file01.txt
-rw-r--r-- 1 root root 0 Dec 21 14:29 file02.txt
-rw-r--r-- 1 root root 0 Dec 21 14:29 file03.txt
[rodel] (local) root@192.168.0.13
```

This causes the results of the **ls** command to display additional details for each file, such as the permissions, group and owner, date and time stamp for each file.

Lab Work:

- 1) Try the **ls** command and see what files are in the current directory (*aka, folder*).
- 2) Create the following files using the **touch** command: (*Hint: see the example above*).
file01.txt
file02.txt
file03.txt
- 3) Try the **ls** command again and you should see the files you just created.
- 4) Lets create some more files, this time 6 files named **HelloAdies-01.txt** through **HelloAdies-06.txt**. But this time we'll learn some super helpful skills to make the process easier.
 - a) **touch HelloAdies-01.txt**
 - b) Press the [Up-Arrow key] once to **recall the last command**
touch HelloAdies-01.txt
 - c) Update the filename to **HelloAdies-02.txt** then press [Enter]
 - d) Try the **ls** command to see if your files were created.
 - e) Lets recall the **ls** command, but this time add a space, hyphen, l:

```
ls -l #You will see :
total 0
-rw-r--r-- 1 root root 0 Dec 27 19:35 HelloAdies-01.txt
-rw-r--r-- 1 root root 0 Dec 27 19:35 HelloAdies-02.txt
```

Linux Commands

- f) **SKILL**: We can create the remaining 4 files using a single **touch** command. The form of this command is:

```
touch <filename1> <filename2> <filename3> <filename4>...etc.
```

Copy and paste the following command.

```
touch HelloAdies-03.txt HelloAdies-04.txt HelloAdies-05.txt  
HelloAdies-06.txt
```

- g) Try the **ls -l** command again and see what files are in the folder now.
h) Copy and paste the 2 commands below, one at a time into your session.

```
touch linux4Adies01.txt linux4Adies02.txt linux4Adies03.txt  
linux4Adies04.txt linux4Adies05.txt
```

```
touch weAreAdaTrained-a.txt weAreAdaTrained-b.txt weAreAdaTrained-c.txt  
weAreAdaTrained-d.txt weAreAdaTrained.e.txt
```

- i) Try the **ls** command again, both with and without the **-l** option and observe the similarities and differences.
- 5) The last thing in this section is to see how the **touch** command can update the date/time stamp of files. We'll do a before-and-after demonstration.

- a) We'll use the **ls -l** command to list out the files that all start with "file0..." Here, we can use a **wildcard character**, '*' asterisk to mean any character after 'file0' followed by ".txt"

```
ls -l file0*.txt  
-rw-r--r--  1 root    root          0 Dec 27 19:35 file01.txt  
-rw-r--r--  1 root    root          0 Dec 27 19:35 file02.txt  
-rw-r--r--  1 root    root          0 Dec 27 19:35 file03.txt
```

- b) Note the date/time stamp on each file ... Dec 27th 19:35.

- c) We can :

- **touch** each file, one at a time, by specifying each file name (e.g. **touch file01.txt**, **touch file02.txt**, **touch file03.txt**) OR
- use a single **touch** command and specify each file OR
- use a **wildcard** character to represent any character. Below is the an example of the wildcard method

Linux Commands

- d) In this example the asterisk after the name `'file'` represents any characters that occur after `'file'`, meaning `file01.txt`, `file02.txt`, `file03.txt`.

```
ls -l file*
```

This means: *list storage using the long format, all files that start with "file" and is followed by characters or numbers.*

Below we can see the before and after effects of the touch command on the date/time stamp.

```
[node1] (local) root@192.168.0.8 ~
$ ls -l file*
-rw-r--r-- 1 root root 0 Dec 27 19:35 file01.txt
-rw-r--r-- 1 root root 0 Dec 27 19:35 file02.txt
-rw-r--r-- 1 root root 0 Dec 27 19:35 file03.txt
[node1] (local) root@192.168.0.8 ~
$ touch file*
[node1] (local) root@192.168.0.8 ~
$ ls -l file*
-rw-r--r-- 1 root root 0 Dec 27 21:26 file01.txt
-rw-r--r-- 1 root root 0 Dec 27 21:26 file02.txt
-rw-r--r-- 1 root root 0 Dec 27 21:26 file03.txt
[node1] (local) root@192.168.0.8 ~
```

Linux Commands

We know the `ls -l` command displays way more useful information than just the plain `ls` command alone.

Q: So, why would we ever use just the `ls` command without any options?

ls command

```
clear
ls
AWS-Digital-Training-Access
AWS-EC2-instances.txt
Adies-AWS-files-S3.txt
Adies-Training
AlfredPowerPack
Applications
Desktop
Documents
Documents-Shared-RPi
Downloads
EKS-Fargate-Instructions.txt
Freenove_Ultimate_Starter_Kit_for_Raspberry_Pi
Hello-Adies-welcome2AWS.txt
```

ls -l command

```
total 143632
-rw-r--r--@ 1 darlayoung staff 1619 Jul 5 2023 AWS-Digital-Training-Access
-rw-r--r--@ 1 darlayoung staff 4296 Dec 2 2023 AWS-EC2-instances.txt
-rw-r--r--@ 1 darlayoung staff 169 Oct 5 2023 Adies-AWS-files-S3.txt
drwxr-xr-x  4 darlayoung staff 128 Nov 21 2023 Adies-Training
-rw-r--r--@ 1 darlayoung staff 179 Jul 5 2023 AlfredPowerPack
-rw-r--r--@ 6 darlayoung staff 192 Dec 9 2023 Applications
-rwx-----@ 13 darlayoung staff 416 Dec 21 21:29 Desktop
-rwx-----@ 47 darlayoung staff 1504 Dec 18 20:12 Documents
drwxrwxrwx@ 6 darlayoung staff 192 Aug 24 09:51 Documents-Shared-RPi
drwx-----+ 53 darlayoung staff 1696 Dec 30 05:54 Downloads
-rw-r--r--@ 1 darlayoung staff 1459 Jan 11 2024 EKS-Fargate-Instructions.txt
drwxr-xr-x@ 12 darlayoung staff 384 Jun 9 2023 Freenove_Ultimate_Starter_Kit_for_Raspberry_Pi
-rw-r--r--@ 1 darlayoung staff 2953 May 22 2024 Hello-Adies-welcome2AWS.txt
-rw-r--r--@ 1 darlayoung staff 879 Nov 21 2023 Key Points.md
-rw-r--r--@ 9 darlayoung staff 288 Nov 19 10:41 Learn
drwx-----@ 110 darlayoung staff 3520 Dec 17 09:04 Library
drwx-----+ 4 darlayoung staff 128 May 25 2023 Movies
drwx-----+ 7 darlayoung staff 224 Sep 25 2023 Music
drwx-----+ 5 darlayoung staff 160 Jun 18 2023 Pictures
-rw-r--r--@ 1 darlayoung staff 40099 Dec 29 20:01 PlayWithDockerSession.log
drwxr-xr-x+ 4 darlayoung staff 128 May 24 2023 Public
-rw-r--r--@ 1 darlayoung staff 556 May 2 2024 RPi-USB-Cam-on-RPi.py
-rw-r--r--@ 1 darlayoung staff 831 Dec 29 20:01 Untitled-2.md
drwxr-xr-x@ 3 darlayoung staff 96 Dec 9 2023 Users
-rw-r--r--@ 1 darlayoung staff 427 Dec 5 2023 aws-cli-choices.xml
-rw-r--r--@ 1 darlayoung staff 676 Dec 29 20:01 board.txt
-rw-r--r--@ 1 darlayoung staff 191 Oct 10 2023 darla-travel-learn
-rw-r--r--@ 1 darlayoung staff 371 Jan 11 2024 defaults.json
-rw-r--r--@ 1 darlayoung staff 106 Nov 23 2023 docker-run-for-neo4j.sh
drwxr-xr-x@ 15 darlayoung staff 480 May 28 2023 getting-started
-rw-r--r--@ 1 darlayoung staff 92 Jun 3 2023 hello-world.html
-rw-r--r--@ 1 darlayoung staff 377 Dec 6 2023 jupyter-notebook.txt
-rw-r--r--@ 1 darlayoung staff 9309976 Oct 26 2023 locatedb.old
```

White arrows show the default sort order which is *alphabetical.*
In the top section, the `ls` command displays files in
- left-most column first, top to bottom
- continuing to second left column, top to bottom, etc.

Looking at the green box on the second left-most column, what happened here?
- We went from "Users" to the next file but it starts with "aws-client..."??
- The alphabetical sort also includes case-sensitivity!
- In this case, a capital 'U' comes before a lower case 'a' !

The `ls` command is useful in showing many more file names at once, which can be handy if you're looking for a particular file.
- But you don't see any details like you do when using the `ls -l` command.
- The `ls -l` command shows way more detail but fewer files

A: If there are a lot of files in a directory.

The the screenshot above there are 50 files, but the `ls -l`, only see a screen full of them (33 files) because the first several have already scrolled out of visibility.

For the most part, though, you will most likely use the long list command in your daily work.

Linux Commands

PRO-Tip: Memorize the sequence “`ls -latr`”

Whenever you sit down at a terminal make this the **first** command you do, just to get yourself grounded, in case you're feeling apprehensive or uncomfortable with the command line.

It's also a good practice because it tends to keep your mind on-track with what you need to do, and restores your confidence in your command line knowledge.

Finally, it shows you which working directory in the file system you are in when you first start working. There are more command like these that we'll cover along the way!

`ls -latr` is most commonly used because it shows all files in...

- (l) long format
- (a) all files including hidden files * (*we'll talk about this below*)
- (t) ordered by timestamp
- (r) and in reverse order

meaning newest files at the top of the list.

Options

Earlier we mentioned that most Linux commands have options that change how the command operates.

- 1) The hyphen tells the operating system that one or more options will be invoked
- 2) Most command options can be used alone or in combinations with other options, as the previous example shows.
- 3) When options are combined using a single hyphen, as above,
 - a) the first option must immediately follow the hyphen (e.g., no space between the hyphen and the option letter/number).
 - b) additional option letters must follow each other with no spaces between them
 - c) option letters can appear in any order
- 4) The following examples all produce the same results:

```
ls -latr
ls -rlat
ls -l -a -t -r
ls -t -lr -a
```

Linux Commands

- 5) Some commands require each option to be followed by a value as in the following for a MySQL database:

```
mysql -u <username> -p <password>
```

These options are product specific, and there are usually instructions indicating how to use them.

- 6) Hidden Files (aka "dot files"): In Linux systems, any file or directory that starts with a period (dot) in the name is not displayed when the ls command is used.

Examples: Hidden files:

```
.bashrc , .bash_history, .config , .ssh ... etc.
```

These files are "hidden" to prevent accidental modification to critical system or configuration files. But you can easily display hidden or 'dot files', use ls -a option (*all files, including hidden files*):

Reading Files

Now that we know **how** to list files what about reading **what's** inside of them?

Linux has 3 commands that we can use to read files: cat, more, less

1. **cat** (from the word concatenate): cat *filename* : displays the contents of a file on the screen.

Example:

```
$ ls -latr
total 24
-rw-rw-r-- 1 root    root          85 Jan 17  2018 .vimrc
-rw-rw-r-- 1 root    root       1865 Jan 17  2018 .inputrc
-rw-rw-r-- 1 root    root         43 Jan 17  2018 .gitconfig
-rw-rw-r-- 1 root    root       240 Dec 20  04:29 .profile~
drwxr-xr-x 2 root    root         61 Dec 20  04:59 .ssh
drwxr-xr-x 1 root    root         92 Jan 18  04:30 ..
-rw----- 1 root    root       879 Jan 18  06:45 .viminfo
drwx----- 1 root    root         55 Jan 18  06:45 .
-rw-rw-r-- 1 root    root       307 Jan 18  06:47 .profile
```

So let's see what each file contains using the cat command.

Try the following commands

```
$ cat .vimrc
```

```
$ cat .inputrc
```

Linux Commands

Let's let Linux do the typing for us:

```
$ cat .g[Tab key]
```

Since there's only one file that begins with `.g` Linux knows you want to cat out the `.gitconfig` file, so it auto-fills the rest of the filename... all you need to do is press [Enter]

Take a look at some of the other files using the cat command.

Pros: quick way to see the contents of a file without a file editor

Cons: long files scroll too quickly to see the first several screens.

2. `more` : displays contents of a file just like cat, but pauses after one screenful of lines, and waits for user to press the Down-Arrow key for the next screen.

Pros: allows user to slowly scroll through long files.

Cons: only scrolls in one direction; you can't go back up

3. `less` : works just like more and waits for user to press the Down-Arrow key for the next screen. But user can scroll back and forth using both Up- and Down-arrow keys.

Pros: allows user full control to scroll forwards and backwards in long files.

Cons: None that I can think of. In this case *"Less truly is more"*

Example:

```
$ cat .inputrc
```

Since this is a long file only the last screenful of lines are displayed, and you'll have to scroll back up through using arrow keys to see the full file.

```
$ more .inputrc
```

`more` displays a screenful of lines at a time, but only allows you to scroll down through the file as many times as needed.

To exit the less session, press [Ctrl] c .

Linux Commands

```
$ less .inputrc
```

Like more, less displays a screenful of lines at a time, but also allows you to scroll up and down through the file as many times as needed.

To exit the less session, press [Ctrl] c .

Linux Commands

SECTION 2

- 1) `cp` (copy) : creates an exact copy the contents of a file:

```
cp <sourceFile> <targetFile>
```

- If the newly created copy is in the **same directory**, then the filename must be different than the source

- If the newly created copy will be in a **different directory**, then the filename can remain the same as the original OR be assigned a new filename as well.

In either case `cp` will start with one file and end up with two files

SKILL : Use the `cp` command to make a backup copy of a critical file **before** you modify it.

- 2) `mv` (move | rename) a file :

```
mv <sourceFile> <targetFile>
```

- If in the same directory, `mv` moves a file from one name to another name

- If moving a file to a different directory, `mv` will move a file from one location to another location

In this case, `mv` can either preserve the filename as is

OR

change the name as well as the location.

- 3) `rm` (remove...delete) :

```
rm <targetFile>
```

Lab Activity:

Scenario: We have a vendor application called Splunk used for monitoring and alerting.

We need update the **splunk.conf** configuration file per vendor recommendation.

- Without this file, the application cannot operate.

- In case something goes wrong:

→ we need to be able to quickly revert back to the original file

→ we want to keep the bad version to analyze later to see what went wrong.

Linux Commands

Setup:

- 1) Create a file from thin air (touch) called splunk.conf
- 2) Make a backup (cp) of the splunk.conf. Let's call it splunk.conf.backup
- 3) Let's make sure our backup file was properly created:
ls -latr
- 4) *** Let's pretend something really went wrong and our application is not working!

 - a) Rename (mv) splunk.conf to splunk.conf.broken
 - b) Rename (mv) the splunk.conf.backup to splunk.conf
- 5) After our Lab Activity is over we can clean up (rm) all of the files so we have an empty directory.

Tasks:

- 1) touch splunk.conf # this just creates a file for us to play with in this class
- 2) cp my [Press the Tab key for code completion] [Space] [Press the Tab key again] .backup

Verify that the original and backup files both exist using ls -latr :

```
$ ls -latr
total 0
drwxr-xr-x  1 root    root    19 Jan 19 03:12 ..
-rw-r--r--  1 root    root     0 Jan 19 03:12 splunk.conf
-rw-r--r--  1 root    root     0 Jan 19 03:13 splunk.conf.backup
```

Now let's suppose we discover a problem and need to revert back to the original config file

- 3) First, let's rename the current splunk.conf to splunk.conf.broken
We want the .broken file to analyze it and see where we went wrong

- 4) At this point **two** things are TRUE
 - a) First, we have the following files:

```
$ ls -latr
total 0
drwxr-xr-x  1 root    root    19 Jan 19 03:12 ..
-rw-r--r--  1 root    root     0 Jan 19 03:12 splunk.conf.broken
-rw-r--r--  1 root    root     0 Jan 19 03:13 splunk.conf.backup
```

- b) **BUT** second, our application is *hard-down*, which means it is completely inoperable! Why? Because we no longer have a .conf file ... just a .backup and a .broken file.
Remember, we said "*Without the [.config] file, the application cannot operate*"

Linux Commands

- 5) To quickly restore service we need to revert our backup file to the configuration file by renaming our .backup file to .conf .

We just need to move (rename) the file `splunk.conf.backup` to just `splunk.conf`

```
$ mv splunk.conf.backup splunk.conf
[node1] (local) root@192.168.0.28 /home/darla
```

One more check to see if we renamed the files properly:

```
$ ls -latr
total 0
drwxr-xr-x  1 root    root          19 Jan 19 03:12 ..
-rw-r--r--  1 root    root          0 Jan 19 03:12 splunk.conf.broken
-rw-r--r--  1 root    root          0 Jan 19 03:13 splunk.conf
```

Finally, a refresh of the splunk session should show that we are back in business!

More Practice:

1. Create 5 files with the touch command
2. Change the file names for each filename
3. Make a copy of the each file (with a different name)
4. Remove (delete) some individual files

Linux Commands

SECTION 3

This section covers how to understand and navigate the Linux file structure. Before jumping into the commands, however, it requires a little background.

We'll introduce a few new commands that help us move around the file system, viewing the directory or file structure, and how to **make** and **delete** directories.

Finally, we'll take a step back to the previous section and use the skills and commands learned in Section 1, but in the context of directories. Some of those commands have additional capabilities when we are working with directories

Knowing and understanding the Linux directory or file structure, combined with the commands in both Sections 1 and 2 is the backbone of a strong Linux understanding, and the key to everything you'll do in Linux afterwards.

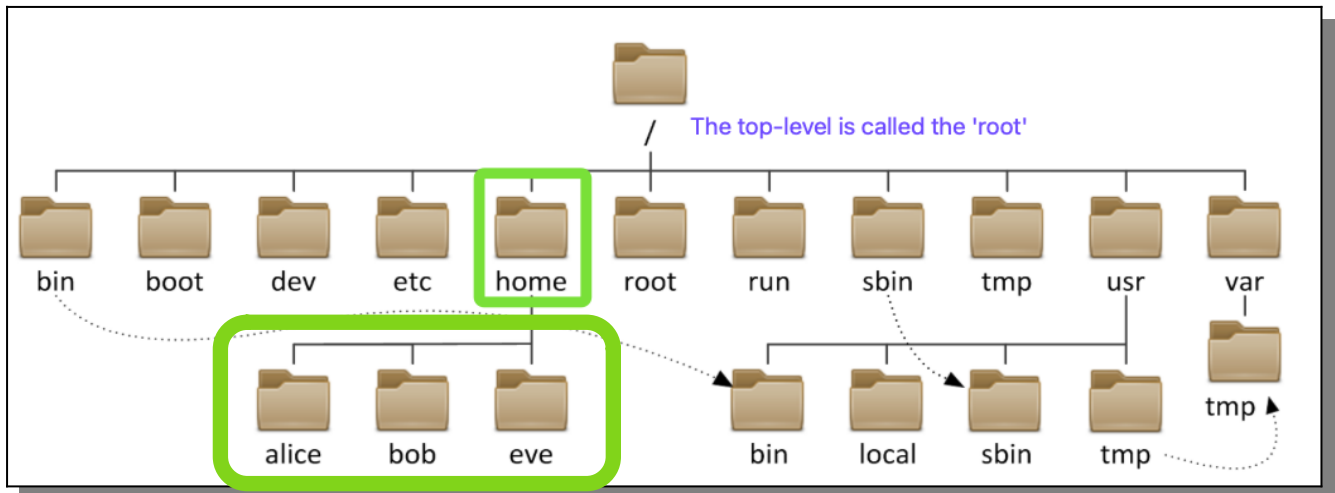
Directories (*aka, folders*), Files, and Paths

Before moving onto the next set of commands we need to first understand how files are organized in Linux.

A diagram is very useful to visualize the basic Linux file structure.

Linux Commands

Often called the *tree structure* because it looks like an upside down tree, Linux is very organized in the way it stores files. It also looks like an organizational hierarchy diagram, and that's essentially what this is, but for files, not people.



Published January 21, 2018 by [Subhash Vadadoriya](http://www.serverkaka.com/2018/01/key-locations-in-linux-file-system-21.html)
<http://www.serverkaka.com/2018/01/key-locations-in-linux-file-system-21.html>

Folder (or directories) can contain files and other folders, called sub-folders or sub-directories. Looking at the diagram above, we see that the `/home` directory contains sub-directories named `alice`, `bob`, and `eve`. Likewise sub-directories can also contain files and even more sub-directories.

Key Points

- 1) The top-level or "*root*" is represented by the forward slash character `" / "`.

Note: many Linux systems also have a directory or folder also named `'root'` reserved for the `'root user'`. The root user on Linux has all all permissions to a system. The `'root user'` should not be confused with `'root'` which is the very top of the file structure (*I know...so confusing, right!?*).

- 2) Below the root are various folders or directories (*these terms are interchangeable*).

Interesting fact: *Before graphical user interfaces like Windows or MacOS existed we used the term "directory". The introduction of the file folder icon in these GUIs crossed over into the command-line world, and is universally understood to mean the same thing in Linux.*

Linux Commands

- 3) Each folder directly under the root can be thought of as having a forward slash before the directory name. And that's actually how it's represented when we type it:

For example,

/home , /dev , /bin, etc. all mean these folders are directly under the root of the file system.

- 4) The /home directory contains a sub-directory for every registered users on most Linux system
- 5) Whenever you log onto a Linux-based systems, your default location will normally be in /home/<your-user_id . For example, in the diagram above, we would reference each sub-directory in the /home folder as:

/home/alice

/home/bob

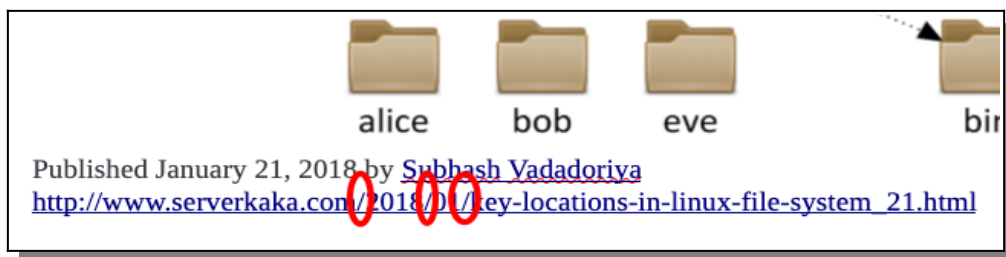
/home/eve

Note: The first "/" in the above diagram means "root" followed by the folder name of home.

The second and any subsequent "/" separate sub-directories and sub-sub-directories from each other. In fact, sub-directories can be nested several levels deep. This is often referred to as the '*file path*', a term which you may encounter in user guides or instructions. For example: /etc/bin/local/splunk.txt tells us where the splunk.txt file is located and how to get to it.

- 6) Believe it or not, you're already familiar with this concept but maybe didn't know it fully.

Consider the diagram on the previous page, more specifically, in the lower-left corner where I cite my reference and the URL reference to the source:



Linux Commands

- 7) This URL is actually **a location for a file** on a Linux server. The “/” characters are separators between sub-directories.

http: Tells us the type of traffic is web traffic

// : Indicates that the following information is a web server

www.serverkaka.com : DNS name of a webserver

/2018 : First sub-directory

/01 : Second sub-directory

/key-locations-in-linux-file-system_21.html : Name of the html file

...still SECTION 3...lol

Next Commands:

pwd, tree, mkdir, rmdir, cd, cd ..

- 1) **pwd** : print-working-directory. The ‘working directory’ is where you are currently located in the file system.
- When first log onto to most Linux systems, your working directory is usually set to `/home/<user-id>`
 - Some systems will include your working directory in the system prompt but most only show you a ‘\$’ sign *aka, the dollar prompt*.

Note about the \$ prompt: This prompt indicates that you have **standard** user privileges which means you are prevented from executing certain commands or accessing protected directories or files.

PRO-Tip: **WARNING** If you ever see the **‘#’** prompt (*aka hashtag or pound-sign prompt*) **stop** what you’re doing immediately and ask a supervisor or coworker.

The ‘#’ prompt means you are operating with **root user privileges!** This is NOT a normal condition and it is possible to do significant and irreparable damage to a system if you are not 100% sure of what you are doing.

Not every system prompt will indicate your location in a file system. To see where you are, use the **pwd** command

Linux Commands

Example:

```
[node1] (local) root@192.168.0.8 ~
$ pwd
/root
[node1] (local) root@192.168.0.8 ~
$ cd /home/d[Tab-key]
darla/      dockremap/
[node1] (local) root@192.168.0.8 ~
$ cd /home/d[A-key][Tab-key]arla/[Enter-key]
[node1] (local) root@192.168.0.8 /home/darla
$ pwd
/home/darla
[node1] (local) root@192.168.0.8 /home/darla
```

- 2) **tree** : (this is probably my most favorite command!) : displays a graphical view of the file structure. It shows directory AND filenames within each directory. **Think of tree like a graphical map of the file structure.**

Examples:

tree by itself assumes you are starting in the current working directory. If my current working directory is my home directory the **tree** shows the following:

```
[node1] (local) root@192.168.0.8 /home/darla
$ tree
.
├── splunk.conf
├── splunk.conf.broken
├── subdirectory-01
├── subdirectory-02
└── subdirectory-03
```

Notice from the display above that it doesn't show the `/home/darla` ; that's because I'm already in `/home/darla`. It only show anything that's in the `darla` subdirectory and below.

You can have **tree** start from any location you specify that's different from your current working directory.

```
[node1] (local) root@192.168.0.8 /home/darla
$ cd / # I changed my current working directory to the root
[node1] (local) root@192.168.0.8 /
$ tree /home # Now I forced tree to start from the /home directory (which is
just below the root). And because I forced the starting point, it printed the /home
and the 2 directories under it
```

Linux Commands

```
/home
├── darla
│   ├── splunk.conf
│   ├── splunk.conf.broken
│   ├── subdirectory-01
│   ├── subdirectory-02
│   └── subdirectory-03
└── dockremap
```

By the way, it really doesn't matter where my current directory is: I could have been several levels deep in some other directory and still shown the same picture *if I specify the starting point* :

Example: Notice where my current working directory is? I'm 4 levels deep in the /sys directory!

```
[node1] (local) root@192.168.0.8 /sys/devices/system/cpu
```

```
$ ls
```

cpu0	cpu2	cpu4	cpu6
cpufreq	isolated	modalias	online
power	smt	vulnerabilities	
cpu1	cpu3	cpu5	cpu7
cpuidle	kernel_max	offline	possible
present	uevent		

PRO-Tip: Use the 'tree' commands "**tree -L 2**" to display the first two levels of in the current location.

- 3) **cd** : stands for **change directory** is probably the next most commonly used command, along with the **ls** command.

cd is how we can navigate the file system.

Example: Suppose my current working directory is `/home/darla` and I want to go to the `/usr/bin` directory.

Linux Commands

The command for this is : `cd /usr/bin`

The **first** '/' means 'root' or top of the file structure. I **ALWAYS have to go through the root to get to another folder**

- `usr` is the name of the directory I want to go into
- the second and any subsequent '/' are separators between directory levels.
- `bin` is the name of the next directory level under `/usr`.

- 4) `cd ..`: By now you've noticed that whenever we list a directory using the `ls -la` (long list, all files) that we see '.' and '..' at the top of the list.

```
$ ls -la
total 16
drwx----- 1 root  root    18 Dec 20 04:59 .
drwxr-xr-x  1 root  root    68 Jan 21 15:34 ..
-rw-rw-r--  1 root  root    43 Jan 17  2018 .gitconfig
-rw-rw-r--  1 root  root  1865 Jan 17  2018 .inputrc
-rw-rw-r--  1 root  root   240 Dec 20 04:29 .profile
drwxr-xr-x  2 root  root    61 Dec 20 04:59 .ssh
-rw-rw-r--  1 root  root    85 Jan 17  2018 .vimrc
```

The single '.' is a shorthand reference to the current directory (we'll see that in action in just a bit).

The double '..' refers to the directory one level above our current working directory,

So, if you want to go up one level, you can use simply use

```
$ cd ..
```

and that will move your current working directory up one level.

This is way faster than typing

```
$ cd /etc/apk every time you wanted to traverse back up.
```

But suppose you wanted to go back up TWO levels. You can use the '/' directory separators to do this:

```
$ cd ../..
```

will move your current working directory up 2 levels.

Linux Commands

PRO-Tip: `cd` just by itself will take you to your `/home` directory.

Example: Suppose my current working directory was in `/etc/apk/keys`

```
etc
├── alpine-release
└── apk
    ├── arch
    └── keys
        ├── alpine-devel@lists.alpinelinux.org-4a6a0840.rsa.pub
        ├── alpine-devel@lists.alpinelinux.org-5243ef4b.rsa.pub
        ├── alpine-devel@lists.alpinelinux.org-5261cecb.rsa.pub
        ├── alpine-devel@lists.alpinelinux.org-6165ee59.rsa.pub
        └── alpine-devel@lists.alpinelinux.org-61666e3f.rsa.pub
```

`$ cd` by itself returns to the `/home` directory.

If you're using the Play-With-Docker Linux session then the home directory is actually `'/root'`

If you're using another Linux system, then normally your home directory would be in `/home/<Your User ID or Name>`

Linux Commands

- 5) `mkdir` : stands for make directory and it's how we create new folders/directories in Linux-based systems.

`mkdir` follows the same rules as other files, meaning they are case-sensitive, and can upper- or lower-case letters, numbers, a hyphen or underscore and a period.

If you begin the directory name with a period (dot), it will be a hidden directory, meaning they won't be displayed with the `ls -l` command, but will be display with `ls -la` option.

Create a directory using the following syntax:

```
mkdir directoryName
```

Example:

`mkdir darlas-files` #this creates a folder called darlas-files in your current working directory

```
[node1] (local) root@192.168.0.28 /home
$ mkdir darla
[node1] (local) root@192.168.0.28 /home
$ ls -l
total 0
drwxr-xr-x    2 root      root           6 Jan 19 23:12 darla
drwxr-sr-x    2 dockremap dockremap      6 Dec 19 07:07 dockremap
```

If you want to create a directory in a different location, specify the directory path and include your new directory name at the end of that path:

Example: Create a sub-directory in the existing `/mnt` sub-directory

```
mkdir /mnt/darlas-files

$ mkdir /mnt/darlas-files
[node1] (local) root@192.168.0.28 ~

$ ls -l /mnt # Check to see if the directory was created
total 0
drwxr-xr-x    2 root      root           6 Jan 19 23:03 darlas-files
```


Linux Commands

Of course the other way to see if your directory was created is to use my favorite **tree** command

```
[node1] (local) root@192.168.0.28 ~
$ tree /mnt
/mnt
├─ darlas-files

2 directories, 0 files
```

6) `rmdir` : stands for 'remove directory' and is used to do just that ... remove or delete directories.

However, the directories MUST BE EMPTY.

Example: Suppose in the last example, in the `/mnt/darlas-files` directory, there were 3 files:

```
$ tree /mnt
/mnt
├─ darlas-files
│   ├── file-01.txt
│   ├── file-02.txt
│   └── file-03.txt
```

Let's delete the `darlas-files` sub-directory

```
$ rmdir /mnt/darlas-files/
rmdir: '/mnt/darlas-files/': Directory not empty
```

We first need to delete all the files in `darlas-files` directory before we can delete the folder itself.

```
$ rm /mnt/darlas-files/file-0*.txt # Notice I used a wild-card here.
```

```
$ tree /mnt $=# tree show us the files are gone...
/mnt
├─ darlas-files
```

Linux Commands

```
$ rmdir /mnt/darlas-files/ # ...and now we can remove the directory
```

```
$ tree /mnt #...finally tree confirms the directory is gone /mnt
```

NOTE: A more common way of doing this is to just use the `rm -r` command which will delete both files AND directories at the same time.

*** USE WITH CAUTION This is where folks get into trouble ***

Example: Same example as before ,, but this time let's use `rm`

```
$ tree /mnt
/mnt
├── darlas-files
│   ├── file-01.txt
│   ├── file-02.txt
│   └── file-03.txt
```

```
$ rm /mnt/darlas-files
rm: '/mnt/darlas-files/': Directory not empty
```

Again, I got the same error!!!

We need the `'-r'` option which means `'recurse'` or repeatedly delete all files in all levels until there is nothing left to delete.

```
$ rm -r /mnt/darlas-files
```

```
$ tree /mnt
/mnt
```

Compared with the `rmdir` command, `rm -r` was much faster and took one less step.

But you can also see how dangerous this could be if you were not careful. Linux does not have an `'Undo'` feature.

Linux Commands

However, `rm` does have 'guard-rail' options: `rm -riv`

where (r) means 'recurse'

(i) lower-case 'i' which means 'inquire' for confirmation

(v) 'verbose' ...show each file or directory being deleted

Example:

```
$ tree /mnt
/mnt
├── darlas-files
│   └── data-files
│       ├── file01.txt
│       ├── file02.txt
│       └── file03.txt
```

```
$ rm -riv /mnt/darlas-files/
rm: descend into directory '/mnt/darlas-files'? y
rm: descend into directory '/mnt/darlas-files/data-files'? y
rm: remove '/mnt/darlas-files/data-files/file01.txt'? y
removed '/mnt/darlas-files/data-files/file01.txt'
rm: remove '/mnt/darlas-files/data-files/file02.txt'? y
removed '/mnt/darlas-files/data-files/file02.txt'
rm: remove '/mnt/darlas-files/data-files/file03.txt'? y
removed '/mnt/darlas-files/data-files/file03.txt'
rm: remove directory '/mnt/darlas-files/data-files'? y
removed directory: '/mnt/darlas-files/data-files'
rm: remove directory '/mnt/darlas-files'? y
removed directory: '/mnt/darlas-files'
```

The guard-rail options specifically ask permission to:

- descend into each directory,
- delete each file, one-by-one,
- and remove each directory

Linux Commands

When completed we'll have just the `/mnt` directory and nothing else in that directory.

```
$ tree /mnt
/mnt
```

So, ...Which is Better? `rmdir` or `rm`?

Short answer, `rm` !

Why? Because, while both `rmdir` and `rm` can recurse sub-directories only `rm` has guard-rail options, `rmdir` does not.

Linux Commands

Lab Activity: LET'S PLAY !!!

- 1) First, let's see where we are in the file system

```
$ pwd
```

If you're using the Play-With-Docker Linux session it may show:

```
/root
```

If you're on your own system it may show:

```
/home/<you're username>
```

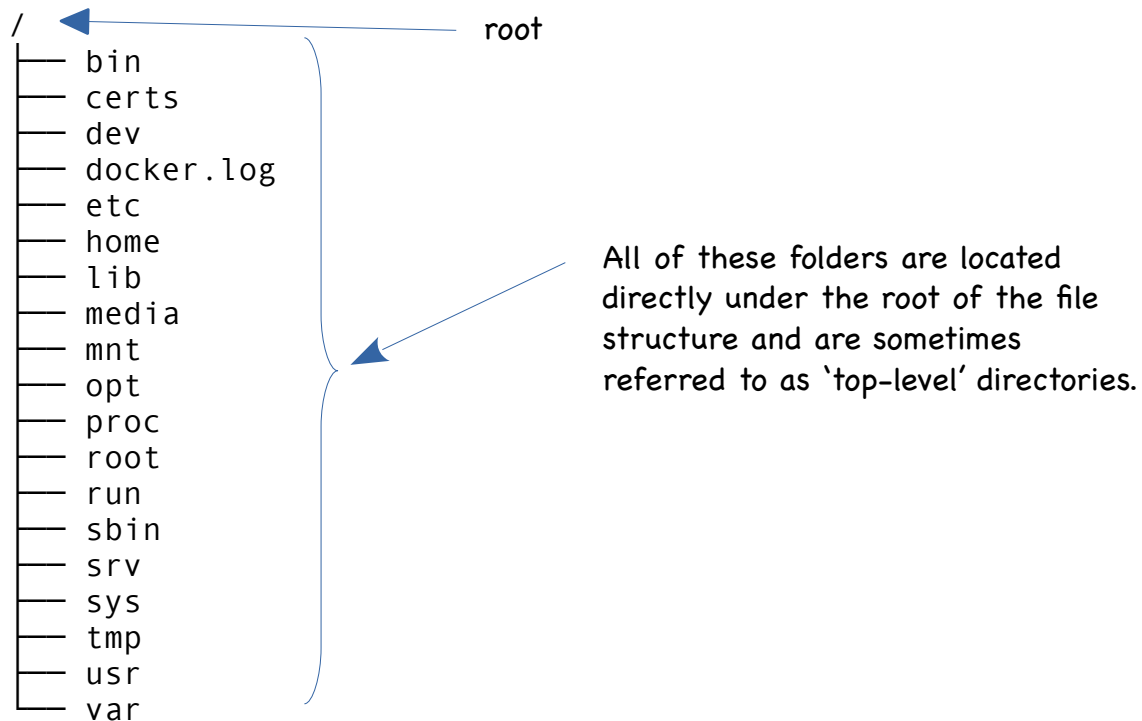
- 2) Let's look at the first level or TOP of our file structure:

```
$ cd /
```

Let's first go to the top of the file structure (*the root*)

```
$ tree -L 1
```

The option "-L 1" (that's upper-case L) means "*only show me the first level of directories*".



Linux Commands

OH! Since we're here...*What would happen if you just run tree without any options???* **WATCH MY SCREEN FIRST.**

So if your screen is going crazy right now, just use **[Ctrl]c** to stop it.

So...um...**What Just Happened???** The `tree` command, by design, lists out every directory and every file in every directory. It also 'recurses' or goes into every sub-directory and sub-sub-directory in the file system!!

To control this, the option `'-L 1'` limits the number of levels deep tree will display.

Try changing the number to 2, 3, 6, etc. and see how the results change when you do that.

Also, to only show directories and not the files, add the `'d'` or `'-d'` option:

```
$ tree -L 2d
```

3) Let's go into the `/bin` folder and see what's there:

```
$ cd /bin    #change to the bin directory
$ pwd        #print the working directory
$ ls -latr   #let's snoop around
```

Let's go to the `/home` directory. If you're using Play-With-Docker, there will only be one folder there for Docker.

Let's create one for ourselves. But first let's make sure where we are:

```
$ pwd
$ /home
$ tree
```

Now that we're sure we don't already have a directory in `/home` let's create one.

```
$ mkdir yourName
$ tree
```

Linux Commands

We should now see both our directory and the one Docker created.

```
$ tree
```

```
.
├── darla
└── dockremap
```

This isn't a printing error or smudge on your glasses. A single '.' is a shorthand symbol that means *current directory* and is discussed below.

Now let's create some sub-directories in our folder#

```
$ cd darla
```

```
$ mkdir folder-01 folder-02 folder-03
```

Verify that your folders were created using either the `ls -latr` command OR the `tree` command

```
$ ls -latr
```

```
total 0
drwxr-xr-x  1 root    root    19 Jan 22 19:53 ..
drwxr-xr-x  2 root    root     6 Jan 22 19:56 folder-03
drwxr-xr-x  2 root    root     6 Jan 22 19:56 folder-02
drwxr-xr-x  2 root    root     6 Jan 22 19:56 folder-01
drwxr-xr-x  5 root    root    57 Jan 22 19:56 .
```

Now let's delete folder-02 using our **guard-rail** options

```
$ rm -riv folder-02
```

```
rm: descend into directory 'folder-02'? y
```

```
rm: remove directory 'folder-02'? y
```

```
removed directory: 'folder-02'
```

Linux Commands

SECTION 4

More VERY useful commands: `history` , `!` , `grep` , `man` , `--help`

1. `history` : Remember how we can use the UP-Arrow and DOWN-Arrows to recall our previous commands? Those are stored in command buffer, and we can look at *all* of our previous commands using the `history` command.

```
$ history
1  tree /bin -L 1
2  cd /home
3  pwd
4  tree -L 3
5  tree
6  mkdir darla
7  tree
8  cd darla/
9  mkdir folder-01 folder-02 folder-03
10 tree
11 ls -latr
12 rm -riv folder-02
13 clear
14 history
```

This command shows a numbered list of every command we've typed, which can be extremely handy in helping us to remember commands, and even the order in which we did something.

The reason each command is numbered is so that we can re-execute them again using the next TIP!

Linux Commands

PRO-Tip: You can re-execute a command from the history command buffer using the `!` exclamation character.

Example: Suppose I want to create the same set of folder as in the previous activity. Remember that we created 3 folders in our home directory, and the deleted one of them

```
$ tree
```

```
.
├── folder-01
└── folder-03
```

3 directories, 0 files

In the history command above, we noticed that Line 9 contained our previously executed make-directory command

```
7 tree
8 cd darla/
9 mkdir folder-01 folder-02 folder-03
10 tree
11 ls -latr
```

To re-execute that command use the exclamation character followed immediately by the line number:

```
$ !9
mkdir folder-01 folder-02 folder-03
mkdir: can't create directory 'folder-01': File exists
mkdir: can't create directory 'folder-03': File exists
```

So it knew the other 2 directories were still there and didn't try to create them, but it DID create the missing one that we deleted earlier.

```
$ tree
```

```
.
├── folder-01
├── folder-02
└── folder-03
```

4 directories, 0 files

Linux Commands

- 2) `grep` : stands for “**g**lobal **r**egular **e**xpression **p**rint” and is used to search for and return patterns within a string value.

In Linux we can take the output of one command and direct it into the input of `grep` to find something specific.

WE use the ‘|’ (pipe) character to tell Linux to do this redirection.

Example 1: Let’s find a specific command from our history buffer using `grep`:

```
$ history | grep 'mkdir'
  6  mkdir darla
  9  mkdir folder-01 folder-02 folder-03
```

We can then use the ‘!’ trick to execute Line 9 without even having to run the history command-line

```
$ !9
```

Example 2: Let’s use `grep` to find the string “bell” in the `.inputrc` file, which is located in `/root .`

There are actually 2 ways to do this.

The first is using the `cat` command and piping the output to `grep`:

```
$ cat .inputrc | grep 'bell'
# do not bell on tab-completion
# set bell-style none
# set bell-style visible
```

OR we can also just use the `grep` command on its own using the form of
`grep 'search-string' filename-to-search`

```
$ grep 'bell' .inputrc
# do not bell on tab-completion
# set bell-style none
# set bell-style visible
```

Linux Commands

- 3) `man` and `--help` : `man` (manual, as in user manual) and `--help` are 2 online help systems included with most Linux systems.

`man` contains a LOT of detail about a command and can be invoke by using the following form:

`man command`

Examples:

```
$ man rmdir
```

```
RMDIR(1)
```

```
General Commands Manual
```

```
RMDIR(1)
```

NAME

```
rmdir - remove directories
```

SYNOPSIS

```
rmdir [-pv] directory ...
```

DESCRIPTION

The `rmdir` utility removes the directory entry specified by each directory argument, provided it is empty.

Arguments are processed in the order given. In order to remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first so the parent directory is empty when `rmdir` tries to remove it.

The following option is available:

`-p` Each directory argument is treated as a pathname of which all components will be removed, if they are empty, starting with the last most component. (See `rm(1)` for fully non-discriminant recursive removal.)

`-v` Be verbose, listing each directory as it is removed.

...-

...-

...etc.

`--help` is the shorter quick help which give a much briefer help, but might not always be available for every command.

Linux Commands

```
$ rmdir --help
```

Usage: rmdir [-p] DIRECTORY...
Remove DIRECTORY if it is empty

-p Include parents
--ignore-fail-on-non-empty

4) **alias** : user-defined key words you can assign to a command to make life easier.

An alias can be letters or words that you define to represent a single or multiple commands. You can then execute these with just a single word,

TO see if any aliases are already defined on your system, just enter the word 'alias' by itself:

```
$ alias  
alias vi='vim'
```

The only alias currently defined in my session is 'vi' which is a reference to the vim editor.

Examples:

'll' Instead of typing `ls -latr` all the time you can define the abbreviation 'll' (two lower-case Ls) to represent that command. Then, just typing 'll' instead of 'ls -latr' .

To declare 'll' as an alias use the following command:

```
$ alias ll='ls -latr'
```

From this point on, until you end your session, just type 'll' to long-list the files in your directory.

'hist' Likewise, if you use the history command a lot for recall a specific commands, you can create an alias that searches the history buffer for a key word.

```
$ alias hist='history | grep '
```

Linux Commands

Now, if you want to search the history buffer for all the times you used the `mkdir` command, just type

```
$ hist mkdir
```

`alias` will translate your request to

```
history | grep mkdir
```

The only drawback to defining aliases at the command line is that when you end your session, the aliases go away, and you'll need to recreate them the every time you log on.

Instead, you can place all of your aliases into a file that is executed whenever you log in.

You can either use a file editor such as `nano` or `vim` to edit an existing file such as the `'.profile'` or `.bash_aliases` in many Linux systems.

For Play-With-Docker, however since these sessions always get reset from scratch, the `.profile` file won't maintain any of your aliases.

You can do the following if you don't yet know how to use `vim` or `nano` editors:

Copy the following commands (and add your own, too) into a file you keep on your local computer.

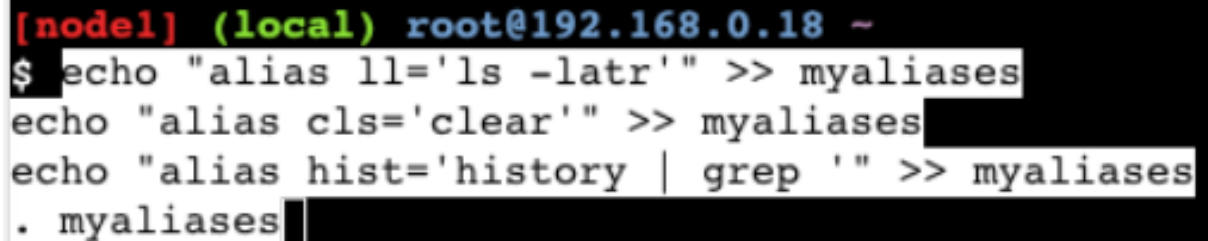
When you start a Play-With-Docker session, just copy/paste these lines into your session when you begin.

```
echo "alias ll='ls -latr'" >> myaliases
echo "alias cls='clear'" >> myaliases
```

Linux Commands

```
echo "alias hist='history | grep '" >> myaliases  
. myaliases
```

In my session after pasting the above lines it waited for me to press [Enter], which then executed each line.

A terminal window screenshot with a black background and white text. The prompt is [node1] (local) root@192.168.0.18 ~. The user enters four commands: \$ echo "alias ll='ls -latr'" >> myaliases, echo "alias cls='clear'" >> myaliases, echo "alias hist='history | grep '" >> myaliases, and . myaliases. The cursor is at the end of the last command.

```
[node1] (local) root@192.168.0.18 ~  
$ echo "alias ll='ls -latr'" >> myaliases  
echo "alias cls='clear'" >> myaliases  
echo "alias hist='history | grep '" >> myaliases  
. myaliases
```

Typing 'll' gave me the long-list of files, and 'cls' cleared my screen

NOTE: On some systems "dotting in" the myaliases file won't work. Instead try
\$ source myaliases

Linux Commands

What these commands do:

Basically we're creating a file called `myaliases` that will contain 3 of our aliases>

a) Anything between the double-quotes will be written to that file using the '>>' redirection characters

NOTE : We use the double '>> myaliases' so that we can APPEND additional lines to this file. A single '>' would overwrite the file every time we added another alias.

b) Double-quotes need to surround the whole alias command so that when we write it to a file it stays together as one unit. Without the double-quotes around the whole command, the echo command may incorrectly interpret it as multiple lines.

c) We then use the echo command to print everything between the double-quotes. Ordinarily it would just be displayed on the screen. But instead each of the 3 commands will be redirected into a file.

d) Last of all, we essentially execute our script by typing a '.' dot followed by a space, and then the script file name.

This will force Linux to execute each line in our script file one at a time.

Again, if you get an error on your system, try this instead:

```
$ source myaliases
```

Linux Commands

SECTION 5

This section covers some more redirection of output, similar to the ‘|’ pipe character that we used with the grep command.

- 1) ‘>’ : The single ‘>’ character directs the output of a command to a file.

If the target file does not exist, the redirect will create it.

If the file already exists, the ‘>’ command will OVERWRITE it with the new contents.

Example: Suppose you want to save the contents of your history buffer to a file for later use:

```
$ history > darlas-command-history.txt
$ cat darlas-command-history.txt

1  cls
2  clear
3  history
4  mkdir -p /home/darlas-files/data-files /home/darlas-files/code-repo
5  touch /home/darlas-files/code-repo/code01.txt /home/darlas-files/code-
repo/code02.txt /home/darlas-files/data-files/data01.txt
/home/darlas-files/data-files/data02.txt
6  tree /home/
7  history > darlas-command-history.txt
```

- 2) ‘>>’ : The double ‘>>’ character does the same thing as the above redirection, but with one difference.

If the target file does not exist, the ‘>>’ redirect will create it.

If the file already exists, the ‘>>’ command will APPEND new data onto the existing file.

The ‘>>’ is often used to create running logs for applications.

- 3) echo : echo is primarily used to print in shell scripts (which are like batch files in Windows CMD or PowerShell) to the screen during execution, like status messages.

Linux Commands

But `echo` is also very useful while debugging system issues because it can print the value of system variables.

In Linux most system variable are by convention almost always UPPER-CASE. To access the value stored in these variables, prefix the variable name with a '\$' sign.

`HOSTNAME` or `HOME` are two such examples.

To print the values stored in these variables on the screen use:

```
$ echo $HOSTNAME
node1
```

```
$ echo $HOME
/home/darla
```

```
$ echo $USER
darla
```

```
$ echo $SHELL
/bin/bash
```

```
$ echo $PATH
/Library/Frameworks/Python.framework/Versions/3.13/bin:/opt/homebrew/bin:/opt/homebrew/sbin:/usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/var/run/ ...
```

PRO-Tip: Did you know you can use 'echo' to create AND populate files? Use `echo "someStringData" > fileName` (include the quotes) and that file will be created with the text you specify.

To create a file with some text in it, use both the 'echo' command and the redirection symbol '>' or '>>'

Example 1: Single '>' redirect means write or *overwrite*

```
$ echo "Practice file made with the echo command" > test-echo.txt
$ cat test-echo.txt
Practice file made with the echo command
```

Linux Commands

```
$ echo "Practice file made with the echo command2" > test-echo.txt
$ cat test-echo.txt
Practice file made with the echo command2
```

Notice the first value was overwritten by the second 'echo' command.

Example 2: Double '>>' redirect means write or *APPEND*

```
$ echo "Practice file made with the echo command" >> test-echo.txt
$ echo "Practice file made with the echo command2" >> test-echo.txt
$ cat test-echo.txt
Practice file made with the echo command
Practice file made with the echo command2
```

The results of both 'echo' commands can now be seen in the same file.

The double '>>' redirect is often used when creating log files where each event is appended onto the same file to show the running progression of a process.

Linux Commands

SECTION 6

This last section describes Linux File Permissions ...all those rwx's when we look at files using the `ls -l` (long-list) option...what do these mean?

```
ls -lh /usr/bin
-rwxr-xr-x    1 root    root    223.7K Jun 22  2023 make
-rwxr-xr-x    1 root    root    306.0K Dec 14  2023 jq
-rwxr-xr-x    1 root    root    154.2K Mar 12  2024 patch
-rwxr-xr-x    1 root    root    228.6K Mar 27  2024 xgettext
```

Short Answer, these rwx's represent the permission each file has, or more accurately, what accesses a person or application must have to access these file.

There are 3 groups of rwx's:

- The first group represents the user's permissions
- The middle group represent the group's permissions
- The last group represents other, meaning anyone else

The rwx's themselves stand for

r - Read
w - Write
x - Execute

So all 4 of the files above have the same pattern of rwx's meaning they have the same permission requirements.

The Owner can Read, Write, and Execute the filename

The Group and the Other can only Read and Execute, but not Write.

By the way, here are what each of the columns mean:

Permissions	Links	User	Group	Size	Date	FileName
-rwxr-xr-x	1	root	root	223.7K	Jun 22 2023	make
-rwxr-xr-x	1	root	root	306.0K	Dec 14 2023	jq
-rwxr-xr-x	1	root	root	154.2K	Mar 12 2024	patch
-rwxr-xr-x	1	root	root	228.6K	Mar 27 2024	xgettext

Linux Commands

NOTE: The leading ‘-’ before the first rwx indicates the file type.

A ‘-’ indicates this is a normal file,
An ‘l’ indicates this is symbolic link
A ‘d’ indicates this is directory

More discussion on this later below.

Normally that would be the end of the discussion, ...except when we run into the situation of not being able to access files or directories because of permissions. It may be up to us to change the permissions of a file or directory so that our application can function properly.

chmod : stands for ‘change mode’. We use 3 digits to represent the permission requirements for a file. Each digit corresponds to the User, the Group, and the Other. If you’re wondering whether the 3 digits mentioned here corresponds to the rwx we just saw above, you’re correct!

So, how does a single digit represent read, write, and execute for the User, another digit represent read, write, and executed for the Group, and the last for the Other?

The answer is binary digits. (Don’t freak out ...this is actually a lot easier than it sounds)

There is a **companion spreadsheet** file that goes with this, which we’ll reference. In that spreadsheet we’ll first explain what binary digits are and then how they are used in file permissions. It is actually kind of ingenious and you may find this method useful in future problem-solving situations.

Please refer to the spreadsheet **Linux-file-permissions1.xlsx**

Then we’ll come back afterwards to see how we use the chmod command to set file permissions on a file.

Linux Commands

To set permission on a file use the chmod command as follows:

chmod <3 digits> <filename>

Examples:

To set permissions on a file to be fully accessible by anyone

chmod 777 myfile.txt

NOTE: This is not recommend for most files because anyone can open, modify, or delete a file. However, it can sometimes be used as a quick troubleshooting tool to determine if an application's problem is permission related or something else.

To set permissions on a file to be read-only by just the user but no one else

chmod 700 myfile.txt

Below are some examples from the spreadsheet file:

- 'chmod' = change mode
- The symbolic command is used to change just one or two permission bit... often seen in instructions to temporarily change permissions.
- The command is used by starting with "chmod" followed by:
 a letter representing the specific permission type (user, group, or other)
 a "+" or a "-" to add or remove a permission
 and then a letter, r, w, or x to specify the type of permission to add or remove.
For example: If we need to temporarily modify my-file.conf to make it writeable (744) and it's usually set for read only (644):
 chmod u+x my-file.conf; (hint: You can verbalize this as "Change mode, User, add eXecute bit")
To revert it afterwards: chmod u-x my-file.conf

(Owner) User	Group	(Everyone) Other	Decimal Value Used in the chmod command	Columns										chmod commands	Verbalize as ...	
				U	G	O	Links	User	Group	Bytes	Month	Day	Time/Year			FileName
rwx	rwx	rwx	777	rw	xr	wx	1	pi	pi	28	Jul	3	20:39	Learn001.py	chmod u+r,u+w,u+x,g+r,g+w,g+x,o+r,o+w,o+x Learn001.py	Change mode, for user add read, add write, add execute, for group add read, add write, add execute, for other add read, add write, add execute.
rwx	r x	r x	755	rw	xr	-x	1	root	root	184	Jul	9	20:55	cmdline.txt	chmod u+r,u+w,u+x,g+r,g+x,o+r,o+x cmdline.txt	Change mode, for user add read, add write, add execute, for group add read, add execute, for other add read, add execute.
rwx	rwx	r x	775	rw	xr	-x	1	root	root	184	Jul	9	20:55	config.txt	What command would you use to grant read-only access to owner a NO access to anyone else for this file?	Change mode, for user add read, add write, add execute, for group add read, add write, add execute, for other add read, add execute.
rw-	r -	r -	644	rw	-r	--	1	root	root	861	Jul	3	20:31	.bash_aliases	What command would you use to grant full access to everyone for this file?	?
r -	r -	r -	444	r--	-r	--	1	root	root	2.2K	Mar	26	2019	python.mk	What command would you use to make this file only executable for Other and NOT readable or writable by anyone else. Hint: --x--x--x	?

If you need to modify permissions on a directory that contains sub-directories specify the '-R' option to recurse sub-directories.

Linux Commands

Revisiting Old Friends...

Now that we've learned the basic commands and even some additional options, let's go back to a couple others.

`mkdir -p` : use the `-p` option to create a directory and any sub-directories in the process:

Example: Suppose I want to create a directory named `darlas-files`, and within that directory I want to have a subdirectory for code files and another one for data files.

If I tried something like `mkdir darlas-files/code-files` I will get an error indicating the directory doesn't exist

```
$ mkdir darls-files/code-files
mkdir: cannot create directory 'darls-files/code-files': No such file or directory
```

This is because the parent-directory, `darlas-files` doesn't exist yet, so it cannot create `code-files`.

Normally I would need to create `darlas-files` first, then descend into that directory in order to create any sub-directories.

But if I use the `-p` option, then I can create both directories at the same time.

```
$ mkdir -p darlas-files/code-files
[node1] (local) root@192.168.0.18 /home
$ tree
```

```
.
├── darlas-files
│   └── code-files
└── dockremap
```

Another helpful tool is `'&&'` double-ampersand.

The double-ampersand is how we can execute 2 commands on the same line.

Suppose I normally enter the following 2 commands together:

```
$ touch myfiles.txt
$ ls -la
```

Linux Commands

A more convenient way to do this

```
$ touch myfiles.txt && ls -la
```

Let's bring this all together.

We've covered a lot of ground and learned a lot of solid skills that make up the bulk what developers and system administrators use on a daily basis when working with Linux!

The real value in what we've learned is when we use all of the command and skills together.

There is no ONE command or skill that will do everything for you, although you may have your favorites. (*My personal favorites are the tree and history commands*)

But there is ONE CONCEPT that having a solid working understanding of will make a majority of the work you do in Linux much easier and more efficient.

File Paths : PATH IS POWER!!!!

Understanding file paths and how to navigate them applies to almost all of the commands that we've learned. To illustrate this let's revisit the commands we've covered, but this time using all the knowledge and skills we've acquired so far.

The following are commands we've learned :

Command	Meaning & Function	Works with File-paths?
ls	List storage	Yes
touch	Touch files to create them or update their timestamp	Yes
cat / more / less	Display the contents of files on screen	Yes
cd / cd ..	Change directory	N/A
mkdir	Make directory	Yes
rmdir / rm	Remove directory	Yes
cp	Copy files and directories	Yes
mv	Rename or move files and directories	Yes
rm	Remove files and directories	Yes
pwd	Print working directory	N/A

Linux Commands

When you first log into a system, you are normally in your home directory

Examples:

```
$ pwd  
/home/<yourUserName>
```

Working Review (we can do this together, or feel free to do on your own for practice).

The challenge is to see how many of the tasks below we can do without ever leaving your home directory.

Use:

- any and all of the
 - commands (ls, touch, cp, mv, rm, mkdir, rmdir, history, grep, etc.) and
 - skills ([Up-Arrow]/[Down-Arrow], [Tab] keys, backup files, etc.) we've learned,
 - whatever you already know, or
 - feel free to look it up in
 - - - help,
 - **manual** pages,
 - Google,
 - ChatGPT,
 - ask each other
- ...basically whatever you need to do (*this is how it's done in the real world*).

PRO-Tip: For this exercise, using the [Tab] key for code completion is an invaluable tool. Aside from not having to type as much, **code completion helps with accuracy**. If you press the [Tab] key expecting to see a file or directory name pop up but it doesn't, then either you're in the wrong location or you've misspelled a name.

From your home directory, and without using the 'cd' command, try to do the following :

- 1) First, go to your default home directory (it doesn't matter if you're in the Play-With-Docker instance or your own system)...i.e., wherever the cd command by itself takes you.
- 2) List the contents of the /usr/bin directory
- 3) In your default home directory, create a zero-byte file named hello.txt and confirm it was created.

Linux Commands

- 4) Create a directory named 'my-directory' within the /home directory
- 5) Copy the file created in Task 3 above into the /home/my-directory
- 6) Use the tree command to show the contents of the /home directory. (*If you don't have the tree command available just use ls -l instead.*)
- 7) Make **two** sub-directories within /home/my-directory with the following names:
my-data-files
my-scripts-files
- 8) Use the tree (or ls) command again to show the contents of the /home directory and see what changed
- 9) Now MOVE the file hello.txt created in Task 2 to the my-script-files folder, and also rename the file hello.sh
Note: in most Linux systems, the extension 'sh' means the file is a 'shell script', which is a file that contains one or more command-line commands that can be executed in the same order every time.
- 10) Once again, use the tree command again to show the contents of the /home directory and see what changed
- 11) Let's put some actual data into the hello.sh file:
 - a) Let's take the content from the history command and put it into the hello.sh file.
 - b) View the hello.sh file and verify that it was populated correctly.

NOTE: You can copy/paste the contents from history on your screen to your own local file to review and study afterwards.

This is especially handy if things didn't work properly.... We can go over it together.

- 12) FINALLY, let's clean up after ourselves.
 - a) Delete the my-directory folder in the /home file using the guard-rail options
 - b) Delete the hello.txt file from the default directory that we created in Task 3.

IF you've made it this far, CONGRATULATIONS! Even if things didn't work out 100% ... you still reached the finish line.

Linux Commands

Practice by Playing:

It's better to learn (*and make mistakes*) when you're not under pressure and you are free to experiment and go down rabbit-holes when researching topics.

- Go back over each task above and try them out often.
- Try variations of the tasks
- Make up your own challenges
- Look up additional Linux training on YouTube, Google, etc.
- Feel free to ask me any questions at darla.teachestech.998@gmail.com

Extras:

7 – FILE PERMISSIONS

File type:
- → regular file
d → directory

Permissions: -rwxrwxr--

Legend:
Execute (x)
Write (w)
Read (r)

PERMISSION	EXAMPLE
U G W	
rwX rwX rwX	chmod 777 filename
rwX rwX r-X	chmod 775 filename
rwX r-X r-X	chmod 755 filename
rw- rw- r--	chmod 664 filename
rw- r-- r--	chmod 644 filename

NOTE: Use 777 sparingly!

LEGEND
U = User
G = Group
W = World

r = Read
w = write
x = execute
- = no access

Linux Commands

For more information about the Linux file structure, Google it or watch some YouTube Videos to learn more about it.

Online Resources:

Linux File Structures Explained: <https://linuxhandbook.com/linux-directory-structure/>

Helpful YouTube Videos / Instructors:

1. Learning Linux TV: Linux Commands for Beginners 01:
https://www.youtube.com/watch?v=lvSoxOMg5_c&list=PLT98CRI2KxKHaKA9-4_I38sLzK134p4GJ
2. Joe Collins -EzeeLinux: Learning the Linux File System:
<https://youtu.be/HIXzJ3Rz9po?si=wunlBpWPilrLrvKO>
3. Tech World with Nana : General Development Concepts (literally one of my favorite instructors!): <https://www.youtube.com/@TechWorldwithNana/featured>
4. Tiny Tech Tutorials : Cloud Computing Tutorials and Hands-on Projects:
https://www.youtube.com/results?search_query=tiny+technical+tutorials