

File System Cheat Sheet

Directory	Purpose
/	The Very Top (Root) of The File Tree. Holds Everything else.
/bin	Stores Common Linux user command bin aries. e.g date, cat, cal commands are in here.
/boot	Bootable linux Kernel and bootloader config files
/dev	Files representing dev ices. tty=terminal, fd=floppydisk, (sd or hd) = harddisks, ram=RAM, cd=CD-ROM
/etc	Administrative Configuration files. The format for many of these configuration can be found in section 5 of the Linux Manual.
/home	Where the home directories for regular users are stored. For example, mine is at /home/ziyad
/media	Unlike /dev, /media is usually where removable media (USB sticks, external hard drives etc.) are mounted.
/lib	Contains shared lib raries needed by applications in /bin and /sbin to boot the system.
/mnt	A place to mount external devices. This can still be used but has been superseded by /media
/misc	A directory used to sometimes automount filesystems on request.
/opt	Directory Structure used to store additional (i.e opt ional) software
/proc	Information about System Resources
/root	The home folder for the root user aka the superuser (similar to the administrator on Windows)
/sbin	Contains administrative commands (bin aries) for the root (super) user.
/tmp	Contains temp orary files used by running applications.
/usr	Contains files pertaining to users that in theory don't change after installation.
/var	Contains directories of var iable data that could be used by various applications. System log files are usually found here.

“Mastering the Linux File System”

Cheat Sheet

1. The Linux File System

The Linux File System follows a **tree-like** structure starting at a base (or root) directory, indicated by the slash (/).

Locations on the file system are indicated using **file paths**.

Paths that start at the **base directory** (/) are known as **absolute paths**.

Paths that start from the **current working directory** of the shell, are known as **relative paths**.

For example both of these examples refer to a file called “file1.txt” in the Documents folder for a user called Sarah. The relative path assumes the shell is in Sarah’s home directory.

Absolute: /home/sarah/Documents/file1.txt

Relative: Documents/file1.txt

For more information about the structure of the Linux file system, [please refer to the File system cheat sheet](#) provided in the resources section of the lecture entitled “The Structure of the Linux File System”

2. Key Commands for Navigating the File System

pwd	Print on standard output the absolute path to the shell’s current working directory.
cd [<new location>]	Change the shell’s current working directory to the optional <new location>. If no location is provided, return to the user’s home directory.
ls [<location>]	List out the contents of the optional <location> directory. If no <location> is provided, print out the contents of the shell’s current working directory.

3. Key Shortcuts when Navigating File System.

~	The current user’s home directory
.	The current folder.
..	The parent directory of the current folder.

4. Wildcards and Regular Expressions

Regular expressions are patterns that can be used to match text. In Linux, they are used to allow a user to make rather generic expressions about what files they want a command to operate on.

Creating regular expressions to match filenames is known as **globbing**.

The regular expression patterns can be made using **special building blocks** known as **wildcards**.

Wildcards are symbols with specific meanings to the shell.

We covered the 3 most common types of wildcard:

*	Matches anything, regardless of length.
?	Matches anything, but for one place only.
[options]	Matches any of the options inside for 1 place only.

These 3 wildcards are incredibly versatile and will cover an exhaustive amount of your regular expression requirements.

For more on wildcards please check out these amazing blog posts ([blog post 1](#) & [blog post 2](#))

5. Creating Files and Directories

touch <file>	Creates an empty file. e.g. touch ~/Desktop/file1.txt
mkdir <directory>	Creates an empty directory. e.g. mkdir ~/newdir

6. Deleting Files and Directories

rm <file>	Remove a file e.g. rm ~/Desktop/file1.txt
rm -r <directory>	Removes a directory. e.g. rm -r ~/newdir
rm -i	Removes in an interactive manner. This is a good safety measure.
rmdir <empty directory>	Only remove empty directories e.g. rmdir ~/emptydir

7. Editing Files with the Nano Editor

Nano is a versatile terminal-based text editor. It comes with a built-in toolbar of various commands and all one needs to know in order to use these options are the meaning of 2 special symbols.

^	This is the CTRL key on your keyboard. For example, ^O is CTRL + O .
M-	This is the “meta” key on your keyboard. Depending on your keyboard layout this may be the ALT, ESC, CMD key. Try it out 😊 Assuming M- is the ALT key, then M-X is ALT + X

Here is a link to the [official documentation for nano](#) if you would like to learn more.

8. The Locate Command

The **locate** command searches a **database** on your file system for the files that match the text (or regular expression) that you provide it as a command line argument.

If results are found, the locate command will return the **absolute path** to all matching files.

For example:

```
locate *.txt
```

will find all files with filenames ending in **.txt** that are registered in the database.

The locate command is fast, but because it relies on a database it can be error prone if the database isn't kept up to date.

Below are some commands to update the database and some reassuring procedures in case one cannot access administrator privileges.

Locate -S	Print information about the database file.
sudo updatedb	Update the database. As the updatedb command is an administrator command, the sudo command is used to run updatedb as the root user (the administrator)
Locate --existing	Check whether a result actually exists before returning it.
locate -limit 5	Limit the output to only show 5 results

9. The Find Command

The **find** command can be used for more **sophisticated search tasks** than the **locate** command. This is made possible due to the many powerful options that the **find** command has.

The first thing to note is that the **find** command will list both files *and* directories, below the point the file tree that it is told to start at.

For example:

```
find .
```

Will list all files and directories below the current working directory (which is denoted by the **.**)

```
find /
```

Will list all files and directories below the base directory (**/**); thereby listing everything on the entire file system!

By default, the **find** command will list everything on the file system below its starting point, to an **infinite depth**.

The search depth can however be limited using the **-maxdepth** option.

For example

```
find / -maxdepth 4
```

Will list everything on the file system below the base directory, provided that it is within **4** levels of the base directory.

There are many other options for the **find** command. Some of the most useful are tabulated below:

-type	Only list items of a certain type . -type f restricts the search to file and -type d restricts the search to directories.
-name "*.txt"	Search for items matching a certain name . This name may contain a regular expression and should be enclosed in double quotes as shown. In this example, the find command will return all items with names ending in .txt .
-iname	Same as -name but uppercase and lowercase do not matter.
-size	Find files based on their size. e.g -size +100k finds files over 100 KiB in size -size -5M finds files less than 5MiB in size. Other units include G for GiB and c for bytes ^{**} .

**** Note:** 1 Kibibyte (KiB) = 1024 bytes. 1 Mebibyte (MiB) = 1024 KiB. 1 Gibibyte = 1024 MiB.

A supremely useful feature of the `find` command is the ability to `execute` another command on each of the results.

For example

```
find /etc -exec cp {} ~/Desktop \;
```

will copy every item below the `/etc` folder on the file system to the `~/Desktop` directory.

Commands are executed on each item using the `-exec` option.

The argument to the `-exec` option is the command you want to execute on each item found by the `find` command.

Commands should be written as they would normally, with `{}` used as a placeholder for the results of the `find` command.

Be sure to terminate the `-exec` option using `\;` (a backslash then a semicolon).

The `-ok` option can also be used, to prompt the user for permission before each action.

This can be tedious for a large number of files, but provides an extra layer of security of a small number of files; especially when doing destructive processes such as deletion.

An example may be:

```
find /etc -ok cp {} ~/Desktop \;
```

10. Viewing File Content

There exist commands to open files and print their contents to standard output. One such example is the `cat` command. Let's say we have a file called `hello.txt` on the Desktop.

By performing:

```
cat ~/Desktop/hello.txt
```

This will print out the contents of `hello.txt` to standard output where it can be viewed or piped to other commands if required.

One such command to pipe to would be the `less` command. The `less` command is known as a "pager" program and excels at allowing a user to page through large amounts of output in a more user-friendly manner than just using the terminal.

An example may be:

```
cat ~/Desktop/hello.txt | less
```

Or more simply:

```
less ~/Desktop/hello.txt
```

By pressing the `q` key, the `less` command can be terminated and control regained over the shell.

Here are some other ways to view file contents:

<code>tac <path/to/file></code>	Print a file's contents to standard output, reversed vertically.
<code>rev <path/to/file></code>	Print a file's content to standard output, reversed horizontally (along rows).
<code>head -n 15 <path/to/file></code>	Read the first 15 lines from a file (10 by default if -n option not provided.)
<code>tail -n 15 <path/to/file></code>	Read the bottom 15 lines from a file (10 by default if -n option not provided.)

11. Sorting Data

A useful ability when working with file data is to be able to sort it either alphabetically or numerically. This behaviour is handled using the `sort` command.

By default, the `sort` command sorts **smallest first**. So if sorting alphabetically, it will by default sort from a – z. If sorting numerically, it will put the smallest numbers first, and largest last.

Here are some common options when using the `sort` command:

<code>sort -r</code>	Reverse the default sorting order.
<code>sort -n</code>	Sort in a numerical manner.
<code>sort -u</code>	Sort data and only return unique entries.

It is also possible to sort tabular data using the `sort` command using one of the columns. This is possible by providing a **KEYDEF** as an argument to the `-k` option.

`sort -k <KEYDEF>`

KEYDEFS are made using a column number and then additional options can be added (without dashes).

As an example:

`sort -k 5nr`

The **KEYDEF** is `5nr`. This will sort using column **5** of the data, and sort numerically (`-n option`) but in reverse (`-r option`).

12. Searching File Contents

The ability to search for and filter out what you want from a file or standard output makes working with the command line a much more efficient process.

The command for this is called the `grep` command.

The `grep` command will return all lines that match the particular piece of text (or regular expression) provided as a search term.

For example:

```
grep hello myfile.txt
```

Will return all lines containing the word “hello” in myfile.txt

and

```
ls /etc | grep *.conf
```

will return all lines with anything ending in “.conf” in data piped from the `ls` command.

Some common options when working with the `grep` command include:

<code>grep -i</code>	Search in a case insensitive manner (upper case and lowercase don't matter).
<code>grep -v</code>	Invert the search. i.e return all lines that DON'T contain a certain search term.
<code>grep -c</code>	Return the number of lines (count) that match a search term rather than the lines themselves.

13. File Archiving and Compression

File archiving and compression is a `two-step` process in Linux.

First one creates a `tarball` to hold all the files that comprise the archive, and then compresses that tarball using one of a variety of `compression` algorithms.

For detailed information on file archiving and compression, please refer to the [File Archiving and Compression cheat sheet](#) provided in the resources section of the lecture entitled “File Archiving and Compression – Part 2”

“ Well done! You should now be well prepared for working with files using the Linux command Line. I hope that this cheat sheet is useful to you if you ever need a refresher! ”

Best wishes, Ziyad.

File Archiving and Compression Cheat Sheet

The Overall Process

Archiving and compressing files in Linux is a two-step process.

1) Create a Tarball

First, you will create what is known as a tar file or “tarball”. A tarball is a way of bundling together the files that you want to archive.

2) Compress the tarball with a compression algorithm

Secondly, you will then compress that tarball with one of a variety of compression algorithms; leaving you with a compressed archive.

1. Creating a Tarball

Tarballs are created using the `tar` command.

Creating a Tar Ball	<code>tar -cvf <name of tarball> <file>...</code>
---------------------	---

The `-c` option: “create”. This allows us to create a tarball. [required]

The `-v` option: “verbose”. This makes tar give us feedback on its progress. [optional]

The `-f` option: Tells tar that the next argument is the name of the tarball. [required]

`<name of tarball>`: The absolute or relative file path to where you want the tarball to be placed; e.g. `~/Desktop/myarchive.tar`. It is recommended that you add `.tar` to your proposed filename for clarity.

`<file>`: The absolute or relative file paths to files that you want to insert into the tarball. You can have as many as you like and wildcards are accepted.

1.1 Checking a Tarball's Contents

Once the tarball has been created, you can check what is inside it using the `tar` command.

Checking the contents of a Tarball	<code>tar -tf <name of tarball></code>
------------------------------------	--

The `-t` option: “test-label”. This allows us to check the contents of a tarball. [required]

The `-f` option: Tells tar that the next argument is the name of the tarball. [required]

`<name of tarball>`: The absolute or relative file path to where you want the tarball to be placed; e.g. `~/Desktop/myarchive.tar`

1.2 [Extracting From a Tar ball](#)

Let's say that you download a tar file from the internet and you want to extract its contents using the command line. How can you do that?

For this you would again use the `tar` command

Extracting a Tar ball's Contents	<code>tar -xvf <name of tarball></code>
----------------------------------	---

The `-x` option: "extract". This allows us to extract a tarball's contents. [required]

The `-v` option: "verbose". This makes tar give us feedback on its progress. [optional]

The `-f` option: Tells tar that the next argument is the name of the tarball. [required]

`<name of tarball>`: The absolute or relative file path to where the tarball is located; e.g. `~/Desktop/myarchive.tar`

Extracting a tarball does **not** empty the tarball. You can extract from a tarball as many times as you want without affecting the tarball's contents.

2. [Compressing Tarballs](#)

Tarballs are just containers for files. They don't by themselves do any compression, but they can be compressed using a variety of compression algorithms

The main types of compression algorithms are `gzip` and `bzip2`.

The `gzip` compression algorithm tends to be faster than `bzip2` but, as a trade-off, `gzip` usually offers less compression.

You can find a comparison of various compression algorithms using [this excellent blog post](#).

2.1 [Compressing and Decompressing with gzip](#)

Compressing with gzip	<code>gzip <name of tarball></code>
Decompressing with gzip	<code>gunzip <name of tarball></code>

When compressing with `gzip`, the file extension `.gz` is automatically added to the `.tar` archive. Therefore, the `gzip` compressed tar archive would, by convention, have the file extension `.tar.gz`

2.2 [Compressing and Decompressing with bzip2](#)

Compressing with bzip2	<code>bzip2 <name of tarball></code>
Decompressing with bzip2	<code>bunzip2 <name of tarball></code>

When compressing with `bzip2`, the file extension `.bz2` is automatically added to the `.tar` archive. Therefore, the `bzip2` compressed tar archive would, by convention, have the file extension `.tar.bz2`

3. Doing it all in one step

Because compressing tar archives is such a common function, it is possible to create a tar archive and compress it all in one step using the tar command. It is also possible to decompress and extract a compressed archive in one step using the tar command too.

To perform compression/decompression using gzip compression algorithm in the tar command, you provide the **z** option in addition to the other options required.

Creating a tarball and compressing via gzip	<code>tar -cvzf <name of tarball> <file>...</code>
Decompressing a tarball and extracting via xzip	<code>tar -xvzf <name of tarball></code>

To perform compression/decompression using bzip2 compression algorithm in the tar command, you provide the **j** option to the other options required.

Creating a tarball and compressing via bzip2	<code>tar -cvjf <name of tarball> <file>...</code>
Decompressing a tarball and extracting via bzip2	<code>tar -xvjf <name of tarball></code>

To perform compression/decompression using the xzip compression algorithm in the tar command, you provide the **J** option to the other options required.

Creating a tarball and compressing via xzip	<code>tar -cvJf <name of tarball> <file>...</code>
Decompressing a tarball and extracting via xzip	<code>tar -xvJf <name of tarball></code>

4. Creating .zip files

Although `.tar.gz` and `.tar.bz2` archives are the archives of choice on Linux, `.zip` archives are common on other operating systems such as Windows and Mac OSX.

In order to create such archives, you can use the following commands.

Creating a .zip archive	<code>zip <name of zipfile> <file>...</code>
Extracting a .zip archive	<code>unzip <name of zipfile></code>


<name of zipfile>: The absolute or relative file path to the `.zip` file e.g. `~/myarchive.zip`

<file>: The absolute or relative file paths to files that you want to insert into the `.zip` file. You can have as many as you like and wildcards are accepted.

“Mastering the Terminal”

Cheat Sheet

1. Opening and Closing the Terminal

Opening The Terminal	CTRL + ALT + T	
Closing The Terminal	CTRL + D	

2. Basic Commands

echo	Prints command line arguments to standard output.
date	Show the current date and time.
cal	Display a calendar.
cat	Stick files together and write joined file to standard output. Good for viewing the contents of 1 file.

3. Command History

history	Show commands previously entered (command history).
!!	Run the previous command.
!50	Run the command that is on line 50 of the output from the history command. (replace “50” as needed).

NB: history -c; history -w; will **clear** the history of commands. 

4. Some Important Definitions

Command	An instruction typed in the terminal and submitted to the shell for interpretation.
Shell	A program that interprets commands for meaning.
Terminal	A graphical window where commands can be typed and submitted to the shell.

5. Command Structure

Each command follows the same overarching structure:

`commandName` `-options` `arguments`



5.1 Command Names

`commandName` must be a valid program on the Shell's Path. To check this, use the `which` command like so:

`which` `commandName`

If a path is returned, then the `commandName` is valid and vice versa.

5.2 Options

You can specify options for each command to customise the commands behaviour. These can be either "short-form" options or "long-form" options.

Each command behaves differently so check the command's manual (`man`) page for the specifics of each command's behaviour.

5.2.1 Short-form Options

Short-form options are where a letter defines an option. Each option is prepended by a dash "-" like so:

`commandName` `-a` `-b` `-c` `args`

To save typing, you could join together the options:

`commandName` `-abc` `args`

Both of these formats are equivalent.

5.2.2 Long-form Options

For some commands, there are long-form options defined to make options easier to identify. Longform options are usually prepended by a double dash "--".

Long-form options **cannot** be joined together like short-form options can.

Whether they are defined or not depends on each specific command, so consult the command's manual page for more information.

If long form options are defined for options "a", "b" and "c", then:

`commandName` `-a` `-b` `-c` `arguments`

is equivalent to

`commandName` `--alpha` `--beta` `--charlie` `arguments`

5.2.3 Command Line Arguments

Command line arguments are a type of input that commands operate on.

Some commands can take an unlimited amount of inputs, some take a specific amount, and some take none at all. Consult the manual page for the specific command for more information.

```
cal 12 2017
```

Here the `cal` command has 2 command line arguments. The number 12 and the number 2017.

5.2.4 Arguments for Options

Sometimes, command options can also take their own arguments (inputs).

```
cal -A 1 -B 1 12 2017
```

Here the `cal` command has 2 options; A and B.

The A option has its own argument (1).

The B option has its own argument (1).

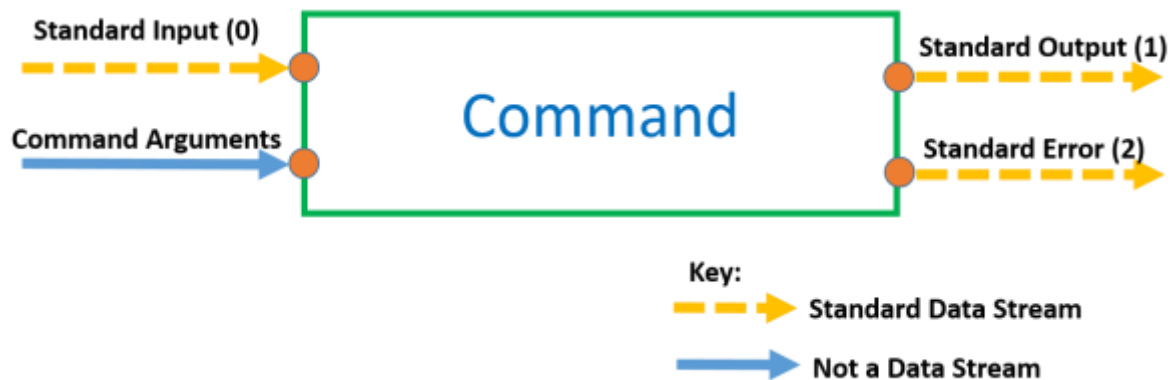
And the `cal` command has 2 command line arguments (12 and 2017).

6. Using the Manual

<code>man -k <search term></code>	Search the manual for pages matching <search term>.
<code>man 5 <page name></code>	Open the man page called <page name> in section 5 of the manual. (replace <page name> and 5 as required)
<code>man <page name></code>	Open the man page called <page name> in section 1 of the manual.

See the Linux manual cheat sheet under the appropriate video for more information about the Linux manual and man pages.

7. Command Input and Output



Standard Data Streams can be **redirected** and are identified using their stream number.

Redirection of the standard output of one command to the standard input of another command is known as **piping**.

7.1 Redirecting Standard Output:

Standard output is stream number **1**. There are 2 methods to redirect standard output.

The *long form*, using the stream number:

```
commandName -options arguments 1> destination
```

Or the *short form*, with no stream number:

```
commandName -options arguments > destination
```

7.2 Redirecting Standard Error:

Standard error is stream number **2**.



Here is how to redirect standard error

```
commandName -options arguments 2> destination
```

Standard error can be redirected at the same time as standard output:

```
commandName -options arguments 1> output_destination 2> error_destination
```

7.3 Redirecting Standard Input:

Standard Input is stream number **0**. There are 2 methods to redirect standard Input.

The *long form*, using the stream number:

```
commandName -options arguments 0< input_source
```

Or the *short form*, with no stream number:

```
commandName -options arguments < input_source
```

8. Piping

Piping is the **connection** of the **standard output** of one command to the **standard input** of another command. Piping using the **pipe character** (**|**) which is accessed by pressing SHIFT + BACKSLASH (\) on most keyboards.

Here is how you would pipe together **commandOne** and **commandTwo**:

```
commandOne -options arguments | commandTwo -options arguments
```

Notice how both commands can have their own options and command line arguments as usual. This piping can go on for as long as is required with as many commands as is required.

8.1 Taking “Snapshots” of pipeline data using the tee command

Redirecting during a pipeline breaks the pipeline.

For example, this **wouldn't** work:

```
commandOne -options arguments > snapshot.txt | commandTwo -options arguments
```

Because redirection is processed by the shell before piping is, snapshot.txt would be created, but this locks up the standard output stream and therefore no data can be passed through the pipeline to **commandTwo**.

NB: *Redirection breaks pipelines*

However, the **tee** command allows us to take a “snapshot” of the data in the pipeline **without** breaking the pipeline.

```
commandOne -options arguments | tee snapshot.txt | commandTwo -options arguments
```

Here, a snapshot of the data coming out of **commandOne** is saved in snapshot.txt, but the data is also successfully piped through to **commandTwo**.

8.2 Piping to commands that only accept command line arguments by using xargs

Piping connects the **standard output** of one command to the **standard input** of another command.

But what if the second command doesn't accept standard input? e.g. the echo command.

The key is to transform the data coming in, into command line arguments.

This is possible using the **xargs** command.

For example, this would **not** work:

```
commandOne -options arguments | echo
```

This **would** work:

```
commandOne -options arguments | xargs echo
```


9. Aliases

Aliases allow you to save your pipelines and commands with easy to remember nicknames so that they can be used later much easier.

You define aliases in your `.bash_aliases` file in your home directory. If it does not exist, you need to create it spelled **exactly** as shown. Note that the preceding period (.) must be included and there should be no file extension (such as `.txt`, or `.pdf`).

Here is how you define an alias in `.bash_aliases`:

```
alias aliasName="THING YOU WANT TO ALIAS"
```

Notice that there are no spaces between the equals sign (=) and the aliasName and the quotes (""). The quotes can be double quote (") or single quotes (').

Let's take an example:

```
alias calmagic="cal -A 1 -B 1 12 2017"
```

With this alias defined in our `.bash_aliases` file, whenever we run the calmagic command it is as if we ran the `cal -A 1 -B 1 12 2017` command.

calmagic is now said to be an **alias** of "`cal -A 1 -B 1 12 2017`".

NB: Aliases may contain either one command or an entire pipeline!

9.1 Piping to an alias

If the **first** command in an alias accepts standard input, then the alias can be piped to; even if it is an entire pipeline!

Our alias is currently:

```
alias calmagic="cal -A 1 -B 1 12 2017"
```

cal is the first command in this alias, but cal doesn't accept standard input.

Therefore, this would **not** work:

```
commandOne -options arguments | calmagic
```

However, if we adjust our alias so that it *can* accept standard input.

```
alias calmagic="xargs cal -A 1 -B 1 12 2017"
```

This will now work:

```
commandOne -options arguments | calmagic
```

And yes, you can pipe out of an alias as well, if the alias produces standard output.

```
commandOne -options arguments | calmagic | commandTwo -options arguments
```

Think of aliases as building blocks that you can use in more sophisticated pipelines.