

The Linux Command Line Bootcamp

CHEATSHEET FOR COLT STEELE'S UDEMY COURSE
(CREATED BY QIUSHI YAN)

- Getting Help

Display the manual page for a command

`man [command] ...`

man pages are a built-in format of documentation. Each man page contains the synopsis of a command syntax. For instance, a simplified synopsis for the **sort** command looks like `sort [-n] [-h] [-k=number] [file] ...`

example man page for **sort**

`[-n]` the -n option is optional

`-k=number` the -k option expects an number

`[file] ...` more than one file can be provided

In summary, **sort** accepts optional argument -n, -h and -k, and -k expects a number, and we can provide more than one file to sort with.

Shortcuts for navigating man pages.

Q	quit man page
B/F	go back/forward a page
/PATTERN	search for a pattern
H	viewing all shortcuts

For shell builtins without a man entry, **help** `[command]` provides instructions.

- Navigation

Command	Meaning
inspect working directory: pwd	
pwd	print working directory
list files of a directory: ls	
ls [dir]	list files of a directory, default to current
ls -a	include dot files
ls -l	use long listing format
ls -h	use human readable sizes
navigate directories: cd	
cd [dir]	change into a directory
cd ..	move up one level
cd /	go to root directory
cd ~	go to home directory
cd -	go to previous directory

- Edit files with nano

`nano file` open file with nano

`nano +line file` open file at a line

nano shortcuts

ctrl+O	write out
ctrl+S	save
ctrl+X	exit nano
ctrl+W	search forward
ctrl+\	replace
M+\\, M+ /	move to the first/last line
ctrl+A, ctrl+E	move to the start/end of a line

.....
Edit /etc/nanorc for further configuration.

- Manipulating Files and Directories

Command	Meaning	
create files: touch		
touch [file] ...	create files	
file [file] ...	print file type	
create directories: mkdir		
mkdir [dir] ...	make directories	
mkdir -p [dir] ...	automatically make parent directories	
copy files and directories: cp		
cp [item1] [item2]	copy a single file or directory item1 to item2	
cp [file] ... [dir]	copy multiple files into a directory	
move and rename files: mv		
mv [item1] [item2]	move or rename the file or directory item1 to item2	
mv [item].. [dir]	move files from one directory to another	
delete files and directories: rm		
rm [item] ...	remove files or empty directories	
Options for rm		
Option	Long	Desc.
-i	--interactive	prompt before removal
-r	--recursive	allow removing non-empty directories
-f	--force	do not prompt

- File Manipulation Cont.

display file contents

Command	Meaning
cat [file] ...	outputs concatenated result of multiple files
less [file]	displays file contents one page at a time
tac [file] ...	prints files in reverse order (last line first)
rev [file] ...	reverse lines characterwise.

cat comes with some handy options

Option	Long	Description
-n	--number	number output lines
-S	--squeeze-black	suppress repeated black lines
-A	--show-all	show non-printable characters such as tabs and line endings

print first / last parts of files inside the current directory

The **head** and **tail** command prints the first/last ten lines of the given file. The number of lines can be adjusted with the **-n** option, or simply **-[number]**.

The **-f** option of **tail** views file contents in real time. This is useful for monitoring log files.

print line, word, byte counts

wc [file] ... prints newline, word, byte counts for each file and a total line of all files

To limit the output, use

- **-w**: print word counts
- **-l**: print line counts
- **-m**: print character counts
- **-C**: print byte counts

Recipe: count total lines of **.js** files

```
wc -l *.js
```

sort lines of files

By default, **sort file** prints each line from the specified file, sorted in alphabetical order. It can also merge multiple files into one sorted whole via **sort file1 file2 ...**.

Options for sort		
Option	Long	Description
-n	--numeric-sort	compare based on string numerical value
-h	--human-numeric-sort	compare based on human readable numbers (e.g., 2k 1G)
-k	--key=KEYDEF	sort via a key
-r	--reverse	sort in reverse order
-u	--unique	sort unique values only

Recipe: find the top 10 biggest files inside a directory

```
ls -lh [dir] | sort -rhk5 | head -10
```


{ The Linux Command Line Bootcamp }

- Redirection and Piping

redirection

A computer program communicates with the environment through the three standard channels: *standard input* (stdin), *standard output* (stdout), *standard error* (stderr)

Redirection Example	
Command	Meaning
standard output to file	
date > file	redirect stdout of date to file, overriding contents
date >> file	append stdout instead of overriding
standard error to file	
cat nonfile 2> error.txt	redirect stderr of cat to file, overriding contents
cat nonfile 2>> error.txt	append stderr instead of overriding
standard input to command	
cat < file	provide file as the standard input for cat
redirect stdout and stdin together	
cat < original.txt > output.txt	provide original.txt to cat, then redirect stdout to output.txt
redirect stdout and stderr together	
ls docs > output.txt 2> error.txt	redirect stdout to output.txt, and if there is an error, redirect error to error.txt
shortcuts	
ls docs > output.txt 2>&1	redirect both stdout and stderr to output.txt
ls docs &> output.txt	redirect both stdout and stderr to output.txt

piping

While redirection operates between commands and files, the pipe operator | passes things between commands, converting stdout of a command to stdin of another command.

Recipe: given a file, transform all letters to lowercase, remove spaces, and save to another file.

```
cat original > tr "[:upper:][:lower:]" | tr -d "[:space:]"> output
```

- Expansion

wildcards and character classes

Shell interprets *wildcard* characters as follows

Wildcard	Meaning
*	any characters
?	any single character
[characters]	any character that's in the set
[!characters]	any character that's not in the set
[[:class:]]	any character included in the class

Common character classes	
[:alnum:]	any alphabetical characters and numerals
[:alpha:]	any alphabetical characters
[:digit:]	any numeral
[:lower:]	any lowercase letter
[:upper:]	any uppercase letter

brace expansion

Brace expansion generates multiple strings based on a pattern.

Syntax	Interpretation
file{1,2,3}	file1, file2, file3
file{1..31}	file1, file2, ..., file30, file31
file{2..10..2}	file2, file4, file6, file8, file10
file{A..E}	fileA, fileB, fileC, fileD, fileE
{a,b,c}{1,2,3}	a1,a2,a3,b1,b2,b3,c1,c2,c3

arithmetic expansion and command substitution

Shell performs arithmetic expansion and command substitution via the `$(expression)` and `${expression}` syntax respectively.

\$(2+2)	4
\$(command)	whatever output command evaluates to

escaping

Quoting let shell treat these special symbols literally. While single quotes suppress all forms of substitution, double quotes preserves the special meaning of \$, \ and ` . Within single quotes, command substitution and arithmetic expansion is still performed.

- Find file by name

the locate command

locate searches pathnames given a substring across the whole computer.

-i	ignore casing
-l=number	limit entries
-e	return update-to-date result (does not use database cache)

the find command

Given a starting point, find lists all files that meets certain option requirement.

find [start_dir] [option] ... [expr]

Options for find		
Option	Example	Meaning
-type	-type d	by file type, e.g., f means files, d means directories
-name	-name '*OLD*'	by file name (pattern specified via wildcards), similar to -path
-size	-size +1G	by file size
-mtime	-mtime -30	by modification time (days), similar options: -ctime, -atime
-exec	-exec rm '{}' ';'	execute custom actions on matched files

We can combine logical operators `-and`, `-or` and `-not` to create complex queries.

Recipe: remove files inside the app folder whose name contains "OLD" or hasn't been modified for more than 7 days

```
find app/ -name '*OLD*' -or -mtime +7 -exec rm '{}' ';' 
```

Recipe: count lines of html and css files in the current directory except the node_modules folder

```
find . -not -path 'node_modules/' \
  \(-name '*.html' -or -name '*.css' \) | xargs wc -l
```

- Search pattern in file contents

the grep command

grep searches for patterns in each file's contents, by default printing each matching line.

grep [option] ... pattern [file] ...

Options for grep	
Option	Meaning
-i	case insensitive matching
-w	matches whole word rather than substring
-r	recursive search, searching the current working directory and any nested directories
-C	count the number of occurrences
-v	select non-matching lines
-l	print matching file names
-C=number	print n lines of matching context
-E	use extended regular expressions.

Unlike find, grep interprets pattern as regular expressions. The basic rules are

Basic regex rules	
.	any single character
^, \$	start or end of a line
[abc]	any character in the set
[^abc]	any character not in the set
*	repeat previous expression 0 or more times

With the `-E` option, we are equipped with additional special characters to write *extended regex*.

Regex	Example	Meaning
?	[abc]?	repeat previous expression 0 or 1 time
+	[abc]+	repeat previous expression multiple times
{n1,n2}	.{2,4}	repeat previous expression a range of times, or exactly n times

Recipe: for all txt files in home directory, search for pattern starts with "console"(case insensitive)

```
find ~ -name '*.txt' | xargs grep -iE '^console.?' 
```


{ The Linux Command Line Bootcamp }

- File Permissions

owners, groups and others

To ensure system security, a permission system is designed dividing users into *owners*, *owner groups* and *others* for each file and directory. Permissions granted to one role won't affect the other two.

reading permissions

The first 10 characters of `ls -l` list permissions for the owner, the group others, e.g.

```
ls -l greet.txt
-rw-rw-r-- 1 colt colt 6 Oct 7 14:34
greet.txt]
```

The first character - indicates the file type, including - (regular file), **d** (directory), **l** (symbolic link) and **c** (character special file). The next 9 characters are permissions for all 3 roles

	Files	Directories
?	can be read	can list contents d
w	can be modified	can create new files, rename files/folders but only if the executable attribute is also set
x	can be executed as a program	allow a directory to be entered or "cd"ed into
-	cannot be read, modified or executed (depending on its location)	cannot show, modify or cd into directory contents (depending on its location)

The above permissions mean `greet.txt` is a regular file, both owners and owner groups can read and modify its content, while others are only permitted to read, no one is allowed execution access.

altering permissions

`chmod [mode] [file]` alters permissions by specifying

- who we are changing permissions for
- will the permission be added or removed
- which permission are we setting

- Permissions Contd.

chmod symbolic and octal notation examples	
u+x	add execution permission to owner
u-x	remove execution permission from owner
+x	add execution permissions for all 3 roles, short for a+x
u+x,go=r	ddd execute permission for the owner and set the permissions for the group and others to read
600	allow read and write access to owner, remove all permissions for groups and others
755	allow read and write for all roles, only allow execution by owners

change identity

Command	Meaning
su - [user]	create a new login shell for the user
sudo -l	see the permitted commands for the user to run as root user
chown [user] [file]	set user the file owner
chown [user]:[group] [file]	set owner and group at once

- Environment

Command	Meaning
printenv	list environment variables
export num=1	define and export variable to child session
alias ll='ls -al'	define custom commands via aliases
PATH="\$PATH:~/bin"	append to the path variable

To persist user-defined environment variables and aliases, we can edit shell startup files such as `~/.bash_profile` (login sessions) and `~/.bashrc` (no-login sessions launched via GUI).

- Basic Bash Scripting

The basic workflow for writing a bash script is

- write script in a file and save it
- make the script executable using `chmod`
- verify shell can find it using `PATH` variable

components of bash scripts

A bash script typically contains a shebang, comments and a series of commands, for example

```
#!/bin/bash
# print a message to the screen
msg='hello world'
echo $msg
```

The shebang `#!/bin/bash` tells OS which interpreter to use when executing, the second line started with `#` are comments that is skipped by shell, any command follows afterwards.

With proper permissions, we can execute the file by `bash [script-path]`. If the path is added to `PATH`, we can call its name directly, e.g.

```
chmod u+x ~/bin/hello
PATH="~/bin:$PATH"
hello
```

- Cron jobs

cron characters

Use `crontab -e` to schedule cron jobs. A job syntax looks like

```
+----- minute (0 - 59)
| +----- hour (0 - 23)
|| +----- day of month (1 - 31)
||| +----- month (1 - 12)
|||| +----- day of week (0 - 6)
|||||
* * * * * command
a b c d e
```

More about cron jobs see course slides and [here](#).

Recipe: run a program at 23:45 every Saturday

```
45 23 * * 6 myscript.sh
```