Command History

```
Up Arrow
CTRL+A   /home   Start of line
CTRL+E   /end End of line
CTRL+R   Command search (type a few chars from command and CTRL+R)
HISTSIZE   Setting History File Size (HISTFILE=0; unset HISTFILE)
```

Moving and renaming Files

```
copy (cp)  CTRL+SHIFT+C -Copy
paste   CTRL+SHIFT+V - Paste
move (mv)
```

Links

```
Symbolic vs. Hard
```

Creating and Deleting

```
touch
mkdir
mkdir -m  umask
mkdir -p  forces creation of parent dirs as needed
rm
rm -r
rm -rf
rmdir
rmdir -p  WARNING: removes current dir and parent dirs
```

man

```
G      moves to end of man page
g      moves to start of man page
/pattern    searches FWD for pattern ie: /logs
?pattern     searches BACK for pattern ie: ?logs
q      quit man
```

less

```
Pg Down; spacebar; CTRL+V; CTRL+F  moves down one screen length
Pg Up; CTRL+B     move back up one screen length
Down Arrow; ENTER; CTRL+N   moves down one line at a time
Up Arrow; CTRL+Y    moves up one line at a time
/pattern     searches FWD for pattern ie: /logs
?pattern     searches BACK for pattern ie: ?logs
/      repeats previous search
q      quit less
man less     for more options
```

locate

```
searches database for linux native commands ie: # locate find
```

whereis

```
searches for binary files, executables, and config files AND man
page page directories [ /usr/share/man/* ];
$whereis man ; $manpath
does not search user directories: /home /* OR various other locations easily
searched by `find' AND `locate' ie: #
```

Process Hierarchy

```
Kernel
   |_____ init - 01
              |_________ BASH
                           |___________ ps
```

ps

```
ps -u Jim   IDs processes use by user: Jim
ps --forest   show complete process tree with parent child relationships
ps -u Jim --forest  show for user: Jim only
ps u U Jim --forest  owned by user: Jim
ps aux   a == shows all procs running on system
   u == users running the proc
   x == CMDs that ran the procs.
```

top

```
IDs procs using most CPU or memory
top -k <PID>  kills process
top -P   sorts by CPU usage
top -M   sorts by Memory usage
```

free

```
Measures Memory usage
- /+ shows actual memory usage by programs (more useful them the Mem: line)
Swap: disk space used when RAM space is not available
```

dmesg

```
Kernel Ring Buffer (log file generated soley in Memory) good for diagnosing driver & hardware issues
dmesg | less
dmesg | grep sda ID kernel messages concerning only 1st hard disk (sda)
```

Regex

```
Wildcards
 ?   matches 0 or more occurances (may or may not occur) +
.   any single character
*  any chars
^  represents start of string
$  represents end of string
[a-zA-Z0-9] can contain any within range [..]
[1..99]  denotes range from 1 to 99
{1,3}   can contain as few or as many of {..,..}
[1-3]*  denotes one or more matches to [1-3]
.*  used to match any single character one or more times
\.   matches the actual dot "."
\?  matches the actual question mark "?"
+  matches 1 or more occurances
(….)  denote sub-expressions
"cat|dog[fish]" can match either cat fish OR dog fish
$touch \{cat,dog,flying,running}fish
$ls -1 | grep "cat|dog[fish]"
|   matches either (….) | (….) essentially an "OR"
```

---

find

```
find -name  denotes case-sensitive file
find -iname   denotes case-insensitive file
find -ilname   " and symbolic linked files
find -inum #####  searches for file w/ inode "#####"
find -size   denotes file of "x" size
find -size 10c  finds file of size exactly 10 bytes
find -size +10M -20M  finds file of size greater than 10MB and less than 20MB
find -size -1G  finds file of size less than 1GB
find -group   denotes belonging to group "name"
find -gid   denotes files belonging to group ID "###"
find -user  denotes files belonging to user: "name"
find -uid   denotes files belonging to user ID: "###"
find -maxdepth  searches to designated depth only for designated files
find /*   starts recursive search at "/"
find /* -type d " for directories ONLY
find /* -type f " for files ONLY
find /* -type p  " for named-pipes ONLY
find /* -type l " for symbolic links ONLY
find /* -type s " for sockets ONLY
find / -type s -exec echo {} 2>/dev/null ; | grep domain* finds socket files
find -name \*.txt  searches from pwd recursively for files ending in ".txt"
find -atime 3   accessed in last 3 days
find -ctime 2  changed 2 days ago
find -mtime 2  modified 2 days ago
find $HOME -mtime 0  searches for files modified in last 24hrs in current-user home-dir
find -cmin -30  changed last 30 minutes
find -amin -60  accessed last hour
find -mmin -60  modified in last hour
find -empty  searches for empty files ONLY
find -executable searches for executable files ONLY
find -exec  execute CMD;true is "0" status returned

find /var/log/ -iname *.log -exec ls -al {} 2>/dev/null \;

             finds all .log files in directory: /var/log/ and lists them all
```

```
find -path  searches in designated path
find -ls  lists files that are found
find -print  print output and automatically adds new-line
find -printf \n prints output followed by new-line
find -prune  searches for files recursively but avoids descending into dirs
find -perm  denotes files with designated permissions level

find / -perm /4000 -uid 0 -ls 2>/dev/null

             finds all SUID files owned by "root" and displays them

find / -perm -2 -type f 2>/dev/null

             finds all GUID world files

find / -perm -2 ! -type -l -ls 2>/dev/null

             finds all GUID world files that are not links

find / ! -path "/proc/*" ! -path "/root/*" -perm -2 -type -f -ls 2>/dev/null

             finds all world files not in directories: /proc OR /root

find / -perm /2 -ls 2>/dev/null

             finds files writable by anyone
```

Cut

```
cut -d: -f1         displays only portion of line before 1st instance of delimiter ``:''
cut -d: -f1-        " and any following strings up to the very next instance ``:''
cut -d: -f1- -s     " " but don't print any lines not containing delimiter ``:''
cut -f3             displays only the 3rd field delimited by space
cut -f2-4           displays only fields 2 through 4 delimited by space
cut -c3-10          displays only the 3rd through 10th characters of each line
```

```
NOTE: Tabs are counted as single spaces on a line; both are counted as single characters
```

Awk

```
awk -F:'{print $1}'         displays 1st field delimited by a ":"
awk'{print $2}'             displays 2nd field, delimited automatically by space
awk"{print $0}'             displays all string data that matches

awk -F: '($3 == 0) {print $1}' /etc/passwd
                            displays 1st field (username) IF the 3rd field (UID) is equal to "0"
awk '{print $NF}'           displays only the last field of every line
```

---

grep

```
grep -c     counts number of lines that match
grep -r ; grep -R   recursive
grep -f     files only
grep -E     regular expressions (GNU)
grep -o     displays ONLY matches of regex / not whole line
grep -i     case-insensitive match
grep -l     displays only file name (full path) of matching regex
```

```
grep -C#     in context of match (# of lines before and after)
grep -A#     displays # of lines after matching regex
grep -B#     displays # of line before matching regex
grep -P     uses Pearl regular expressions (non-GNU)
grep ""     shell quoting (BASH interpretable)
grep ''     shell quoting (NOT BASH interpretable)
```

Command Chaining Operators:

```
&   --Sends process to background (so we can run multiple process parallel)
;   --Run multiple commands in one run, sequentially.
 \  --To type larger command in multiple lines
&&  --Logical AND operator
||  --Logical OR operator
!   --NOT operator
|   -- PIPE operator
{}  --Command combination operator.
    [ -f 99.txt ] || { echo " does not exist"; touch 99.txt; }
()  --Precedence operator
```

Conditionals

```
-e          file exists ?
-f          file exists, and is regular file ?
-d          file exists, and is a directory ?
-s          file exists, and is NOT empty ?
-x          file exists, and IS executable ?
-w          file exists, and is writable by me ?
-gt >       is greater than
-lt <       is less than
-ge >=      is greater than or equal to
-le <=      is less than or equal to
-eq ==      is equal to
-ne !=      is NOT equal to
```

Bash Script using conditional " -f " :

```
shell_env                       #!/bin/bash
[Conditional]                   if [[ -f /etc/password ]]; then
Command                             echo "/etc/password file exists";
[Conditional]                   elif [[ ! -f /etc/password]]; then
Command                             echo "/etc/password file doesn't exists";
                                fi
```

Bash Script using conditional " -w " :

```
shell_env                       #!/bin/bash
[Conditional]                   if [[ -w /etc/passwd ]]; then
Command                             echo "/etc/passwd is writable by me";
[Conditional]                   elif [[ ! -w /etc/passwd ]]; then
Command                             echo "/etc/passwd file isn't writable by me";
                                fi
```

Bash Script using conditional " BOOLEAN " :

```
$(( )) expands "C" arithmetic expressions, but instead of returning a BOOLEAN value, it
returns the math.

shell_env                       #!/bin/bash
[Conditional]                   if [[ 3129 == $(( 15645/5 )) ]]; then
Command                             echo "math checks out";
[Conditional]                   elif [[ 3129 != $(( 15645/5 )) ]]; then
Command                             echo "math is no good";
                                fi
```

When to use (== vs. -eq) & (!= vs -ne) :

```
when comparing a string you should never use -eq OR -ne, instead use == OR !=
when comparing a string integer either is fine to use .. ex:

if [[ banana == apple ]]; then
echo "banana IS an apple"
else
echo "banana is NOT an apple";
fi
= banana is NOT an apple

if [[ banana -eq apple ]]; then
echo "banana IS an apple";
else
echo "banana is NOT an apple";
fi
= banana IS an apple

if [[ banana != apple ]]; then
echo "banana is NOT an apple";
else
echo "banana IS an apple";
fi
= banana is NOT an apple

if [[ banana -ne apple ]]; then
echo "banana is NOT an apple ";
else
echo "banana IS an apple";
fi
= banana IS an apple

if [[ 41 -eq 42 ]]; then
echo "41 is 42";
else
echo "41 is NOT 42";
fi
=41 is NOT 42

if [[ 41 == 42 ]]; then
echo "41 is 42";
else
echo"41 is NOT 42";
fi
=41 is NOT 42

if [[ 41 -ne 42 ]]; then
echo "41 is NOT 42";
else
echo "41 is 42";
fi
=41 is NOT 42
```

```
if [[ 41 != 42 ]];then
echo "41 is NOT 42";
else
echo "41 is 42";
fi
=41 is NOT 42
```

Aliases

```
alias rm='rm -i'                    creates alias to confirm removal
alias vim='nano'                    creates an alias for `nano'
alias gedit='nano'                  "
alias vi='nano'                     "
alias x='cat etc/passwd'            creates an alias for Command: cat/etc/passwd
alias y=$(cat /etc/shadow)          " cat /etc/shadow
alias ls='ls -al'                   creates an alias causing 'ls -al' to be run when 'ls' is used
\ls                                 negates the alias function, so we can run 'ls' without '-al'
alias -p                            view all aliases set (local and global)
unalias ls                          unaliases ls so it no longer resolves to 'ls -al'
```

Redirection Operators:

```
>                                   creates new file with STDOUT; if file exists, it is overwritten
>>                                  appends STDOUT to existing file; if file doesn't exists, it is created
2>                                  creates new file with STDERR; if file exists, it is overwritten
2>>                                 appends STDERR to existing file; if file doesn't exists, it is created
&>                                  creates new file with STDOUT+STDERR; if file exists, it is overwritten
<                                   use contents of file as STDIN
<<                                  accepts ASCII text as STDIN on the following lines
<>                                  specified file is used as both STDIN and STDOUT
```

Sed

```
sed '/.dll/{x;p;p;p;x}' -i document.txt
    directly alters document.txt by adding 3 empty lines before the designated regex (.dll)

sed '/stuff/G;/stuff/G' -i document.txt
    directly alters document.txt by adding 2 empty lines after the designated regex (stuff)

sed -i -e 's/ANCHOVIES/SAUSAGE/g' pizaster.htm
            replaces every instance of "ANCHOVIES" with "SAUSAGE" on pizaster.htm

sed -i -e 's/ANCHOVIES//g' pizaster.htm
            removes every instance of "ANCHOVIES" on pizaster.htm

sed -i '/^#/d' /etc/hosts.allow
            removes all lines starting with "#" from file /etc/hosts.allow
```

Command Substitution

```
A=$(Command)            A=$(cat /etc/passwd)
`Command`               `cat /etc/passwd`
```

BASH Variables

```
A=value                 A=120
B=value                 B=20
expr $A - $B            =100
C=$(expr $A - $B)
echo $C                 =100
D=.mp3                  for x in $(cat songs); do sed -i "s/$/$D/"; done
                        =appends .mp3 to the end of every song in file: songs
```

Sort

```
sort                sorts content according to position on the ASCII table
sort -n             sorts content numerically
sort -u             sorts content uniqely
sort -nr            sorts content numerically reversed
sort -t +
sort -k 2,4         sorts 1st by content in the 2nd column, then by content in the 4th column
```

Uniq

```
uniq                sorts content uniqely
uniq -c             sorts content uniqely with a count reading
```

Script Argument Representation

```
$0 is always the script being executed:

$ ./script.sh content.txt file
        |       |           |
       $0       $1          $2

$1-$9           are arguments that follow the command
${10}-${99}     double-digit arguments
$#              represents number of arguments

usage:          if [[ "$#" != "3" ]]; then
                    echo "usage: $0 content.txt file"
```

Return Codes

```
0 normal return code for successful execution
1 return code for unsuccessful execution
$? variable that represents the return code

usage:          if [[ "$?" != "0" ]]; then
                    echo "something went wrong"
```

passwd

```
passwd          allows you to change user's password
passwd -a       report password status on all accounts
passwd -d       deletes a user's password /"NP" status
passwd -l       locks a user's password /"L" status
passwd -u       unlocks a user's password
passwd -S       gets password status for designated user
```

id

```
id              displays all user and group data for chosen user
id -G           displays other Groups the user belongs to
id -g           displays the GID
id -u           displays the UID
id -n           displays the name associated with UID or GID like below:
id -un          displays user's name id -gn displays user's primary group name
```

Arrays

```
array=({ra..nge})
array=({A..Z})
array=({1..10})

ex:
array=({A..Z})
for x in ${array[@]}; do
    echo $x;
done

ex:
first=({1,3,5,7})
second=({2,4,6,8})
for x in ${first[@]}; do
    for y in ${second[@]}; do
        echo $x $y
    done;
done
```

Case Satements

```
#!/bin/bash
echo -n "Do you agree? [yes or no]: "
read RESPONSE
case $RESPONSE in
    [yY]|[yY][eE][s])
        echo "You answered: yes."
        ;;
    [nN]|[nN][oO])
        echo "You answered: no."
        exit 1
        ;;
    *)  echo "Invalid Answer."
        ;;
esac
```

Functions ()

```
myfunc() {
    commands
}
folder() {
    if [ ! -s $1 ]; then
        mkdir $1
    else
        echo "Whoa Partner!"
    fi
}

ex:
$ folder /etc
Whoa Partner!
```

Pipes Example:

```
$mkfifo fifo
$ls -1 > fifo
```

If you watch closely, you'll notice that the first command you run appears to hang. This happens because the other end of the pipe is not yet connected, and so the kernel suspends the first process until the second process opens the pipe.

```
$cat < fifo
```

Pipes allow totally unrelated programs to communicate with each other.

```
echo -n x | cat - fifo1 > fifo2 &
```

This first command, echos x(without newline) through a pipe into the cat command, BUT cat also has to cat fifo1 into fifo2; the "-"" here is STDOUT, so what happens, is that STDOUT from the (echo -n x |) cmd is included as STDIN / ARG1 for the cat cmd, as is fifo1; essentially placing both on the same stream; this sends "x" into fifo1, which in turn sends "x" to fifo2.

```
cat <fifo1 > fifo1 &
```

This next command string cat's content of fifo2 into fifo1.

If you run the $top cmd, you'll see cat, because it is sending "x" from fifo1 to fifo2 over and over in a loop. Confirm this is happening by $jobs