# Bash Shell

- Cheat Sheet
  - Bash Cheatsheet
  - Vim Cheatsheet

- Disk Usage utility
- Difference between single (' ') quote vs double (" ") quote
- Backslash Escapes
- find glob expansion
- Echo Clear Previous Line
- Diff Command Line Tool
- grep, awk, sed family tool
- Command substitution vs Process substitution
- tr and cut command
- Quoting and Not Quoting Arguments
- How does Program Execute in Linux

# Cheat Sheet

Cheat Sheet

# Bash Cheatsheet

## Multi-line Command

To execute multi-line command in bash script simple put a \ after breaking up your commands. There should be no white spaces after the backslash, else it will fail!

Every single line should be followed by a \ until you finished typing the command.

```
curl https://cat.png \#no spaces after!
-o here.txt \
```

## If-Statements

In Bash the if-statements follows the structure:

```
if CONDITION
then
COMMAND
elif CONDITION
then
COMMAND
else
COMMAND
fi
```

As you can see every if chain will end with *fi* keyword. Every *if/elif* condition statement will be followed by the *then* keyword.

If you want to put the *then* on the same line with the *if* keyword, and because *if, then, else, elif, fi* are all shell keyword they cannot be used on the same-line. To fix this you have to put a ; to end the previous statement and the keyword before you can use another keyword. As an example:

```
if CONDITION; then
COMMAND
fi
```

# Conditions

Okay, now we know the basic of if-statements how do I put it into use by filling in the conditions? There are couple different ways of writing conditions, here I will only go over the most commonly used ones.

1.

```
# The first way is using test
if test <expressions>; then
COMMAND
fi
```

Using this method you can test whether the files exists, and compare values. It has it's own set of syntax for example to check if a certain file named "foo.txt" exists then you would type `test -f "foo.txt"` and it will evaluate to true if only the file "foo.txt" exists.

For comparing values we cannot use the symbols directly, ==, <, >, <=, or >=. Instead we have to use their flag equivalent below:

| Compartor | Flag Equivalent |
|---|---|
| == | -eq |
| > | -gt |
| < | -lt |
| >= | -ge |
| <= | -le |
| or | \|\| |
| and | && |
| not | ! |

You can also append *!* to negate the expression to check the opposite.

**Use = to do string equality comparison.**

2.

```
# The second way is using [] square brackets
if [ some test ]; then
```

```
COMMAND
fi
```

The square bracket is like *test* and essentially all the operators that you can use with *test* you can also use in the square brackets.

There must be a space between the test and the left bracket, [ and the right bracket, ], otherwise Bash cannot understand it!

3.

```
# The third way is using [[]] double square brackets
if [[ some test ]]; then
COMMAND
fi
```

The double square brackets is like an upgrade of the normal square bracket. It comes from *ksh*.

With the double square brackets you can use some of the comparison operators without using the flags. So we are allowed to use >, =, and <, but they are used in **lexicographical comparison!** However, <= and >= still requires the flag equivalent.

4.

```
if command;then
COMMAND
fi
```

Bash runs the command you have provided and then will run the if-statement according to the exit code. It will run it if the exit code is 0, and will not run it if it is not 0.

> Remember in programming, 0 represent the command carried out successfully, and anything not 0 represent some sort of errors occurred.

# Variables & Arrays

To declare a variable in Bash follow the following syntax structure:

```
VAR_NAME=VALUE
```

You must not use any spaces between the variable name and the value! Otherwise, it will error out because you are not following Bash syntax!

# Command Output -> Variable

To store the output of a command into a variable follow the following syntax structure

```
VAR_NAME=$(COMMAND ARGS)


OR


VAR_NAME=`COMMAND ARGS`
```

This will do command substitution, it will execute the command and then substitute the return value as the value.

To prevent the output of the command from being processed for word splitting (i.e. the \n loses their meaning in a text file), you would quote the command substitution `"$(COMMAND ARGS)"` to prevent word splitting.

From the GNU shell specification: **"The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting."**

# Arrays

To declare a array follow the following syntax structure:

```
arr=(1 2 3 4 5 6 7)


OR


arr=(
"hello"
    "world"
    "hehe"
    "xd"
)
```

The array can contain different types, either an integer, float, or even strings.

You can refers to each individual elements using `${var_name[index]}`, the *index* are 1-based indices.

To refer to the entire array for say the usage of running a command with arguments from the entire array you can use `$array_name[@]` to refer to all of the elements from the array.

## Array Operation Summary Table

| Syntax | What it Does |
| --- | --- |
| `arr=()` | Create an empty array |
| `arr=(1 2 3)` | Initialize an array |
| `${arr[2]}` | Get the third element |
| `${arr[@]}` | Get all of the elements |
| `${!arr[@]}` | Get the indices of all the elements |
| `${#arr[@]}` | Get the length of the array |
| `arr[0]=3` | Overwrite the first element with value 0 |
| `arr+=(4)` or `arr+=($another_variable)` | Add a new value to the array |

# Let and (( )) Construct

The `let` built-in command allows you to do arithmetic operations on variables. It can be used to do a simple increment operations. Examples below:

| Command | What it Does |
| --- | --- |
| `let a=11` | Same as `a=11` |
| `let a=a+5` or `let "a = a + 5"` | Both set `a` to be 5 more of itself |
| `let "a <<= 3"` | Left-shifts `a` 3 places |
| `let "a += 4"` or the other math operators | Same as `let "a /= 4"` |
| `let a++` or `let "a++"` | C-style operators works as well! |

Some simple operations like increment cannot be carried out by the Bash directly so you would use `let` command to actually do the increments! You do not need to refer to the variable names using dollar signs.

You can replace all the operators described above in between `(( ))` to have the same effect, minus the need for using double quotations. Addition, subtraction, division, multiplication, bit shifts, post/pre increment all works.

# For Loops

To do for loops there are couples of ways:

1. Looping through array elements

```
for ele in ${array[@]}; do
    echo $ele
done
```

2. Looping through array indices

```
for i in ${!array[@]}; do
    echo $i
done
```

3. Looping through range, the end is included

```
for value in {start..end..step}
do
    COMMAND
done
```

The keyword `break` and `continue` are also available for use in the for loop just like how it would work in any other languages.

# While Loops

While loops have the basic structure as follows:

```
while [ some test ]
do
    COMMANDS
done
```

# Bash Functions

There are two ways of writing a bash function

```
function_name() {
    echo "This is the body of the bash function"
}
```

```
function another_bash_function() {
    echo "This is another function!"
}
```

With either way you would be invoking the function by just calling it like it is a command. For example, to invoke the first function you would simply type `function_name` and then provide any argument that you would like to pass into the function. The arguments passed into the function can be accessed using `$1, $2, $3, ...` just like in bash script. `$#, $@` works as well with respective to the argument passed to the function, not the script!

Cheat Sheet

# Vim Cheatsheet

## Mandatory get out of Vim joke

Ughhh how do I escape Vim?

```
# <ESC> :wq, write the changes to the file and quit
# <ESC> :q, quit if there is no changes
# <ESC> :q!, quit without saving
```

## Basics

```
            ^
            k              Hint:  The h key is at the left and moves left.
       < h      l >               The l key is at the right and moves right.
            j                     The j key looks like a down arrow.
            v
```

This is how you would be moving your cursors around in the file

They can be preceeded by a number to tell how many lines to say go down or go to the left.

For example: Pressing 10 and then h will move your cursors to the left 10 characters.

| **Command** | **Description** |
| --- | --- |
| `x` | Delete the character at the cursor |
| `i, a, A` | Insert character at the cursor, insert character after the cursor (append), and append to the end of the line respectively. |
| `<ESC>` | Put you into normal mode if you're in insertion mode, replace mode, or other modes |
| `w, e, b` | Use it to traverse skip through each word to the start of each other, or to the end of each word. Use `b` to go back a word. |

| | |
|---|---|
| `dw, de, d$, dd` | Delete from cursor up to the next word<br>Delete from cursor up to the end of the word only<br>Delete from cursor to the end of the line<br>Delete the entire line regardless where your cursor is |
| `operator [number] motion` | Operator such as `d` can be specified together with a number to tell it how many times to repeat the motion.<br>Motion tells which text to operate on, `w, e, $` |
| `u, U, CTRL-R` | Undo previous action<br>Undo all the changes on the current line<br>Redo, (undo the undo) |
| `p` | Paste what was deleted with the `d` operator |
| `rx, R` | Replace the current character with x, where x can be any character<br>Enter in replace mode where every character you type will be replacing the one that's on the cursor. |
| `CTRL-G` | Display current line number |
| `G, <NUMBER> G` | Move to the end of the file<br>Move to the line number specified |
| `gg` | Move to the first line of the file |
| `/, ?`<br>`n, N`<br>`CTRL-o, CTRL-i` | Used for searching patters forward and backward respectively<br>Used to find the next occurrence of the pattern in the direction specified, `N` is used to find the previous occurrence.<br>`CTRL-o, CTRL-i` take you back to the older/newer position so you don't have to scroll back to where you are respectively |
| `%` | Used to find the matching ( ), [ ], { } |

| | |
|---|---|
| `:!command` | Will execute external command, like `ls` |
| `:w FILENAME` | Write the current Vim file to another file called FILENAME |
| `v` | Visual mode, let you see what you are highlighting and then you can apply other operator such as deleting the highlighted text. |
| `:r FILENAME` | Read in the specified file and then paste it below the cursor<br>Can be used together with command to paste the output of the command. `:r !ls` |

## Some more command

| **Command** | **Description** |
|---|---|
| `o, O` | To insert a empty line below and above the current line |
| `y` | Copies text, `yy` copies the entire line. Can be used together with `v` to highlight a good section of the file and then paste it with `p` |
| `CTRL-W CTRL-W` | To jump to another opened window |
| `:terminal` | Open up a terminal within Vim this is pretty cool |

## Set command

Some settings in Vim can be changed for example if you want to show line numbers you can do `:set number` to toggle on displaying the line number on the side.

You can toggle it off by prepending a `no` to the original toggle, for example `:set nonumber` will disable displaying the line number on the side.

# Disk Usage utility

## df - File System Disk Usage

The command `df` reports the total disk usage on the mounted file system. For example, how much space is still available in the root partition, how much space is still left on the `/dev/sda1` partition that you have created and where it is mounted at.

By default, the output of `df` list out long numbers can be difficult to interpret, so you can provide the `-h` flag to make the output more human readable, in terms of bigger units essentially.

## du - Disk Usage

This command tells us the amount of spaces that's occupied in the current directory by each of the folders, and how much spaces is occupied by current folder in total. Use `-h` for again human readable format.

If there are nested folders, each of the nested folder's information will also be shown, but you only add the size from the most outer folder that contains them!

You can provide a folder path or a path to a file to see how much space they take up.

Use the `-s` flag to just get a summary of the current folder to see how much space it is taking up.

### Folder Size

A note on the folder size not the content that's within the folder. A folder is essentially another file that contains meta data about all the other files that's contained in it. The size of a folder is typically 4KB in Linux system. So if you just run `du` in a empty folder you should see the total space occupied is 4KB because the current folder takes up 4KB by itself!

## lsblk - List Block Devices

Print out information about block devices like file systems that's mounted or not mounted.

Useful for checking what drives are connected.

# Difference between single (' ') quote vs double (" ") quote

## Single Quote

Using single quote in bash script will preserve the literal value of each characters in the single quotes. It will not do any variable interpolation but instead treat it as it is. Basically what you entered is what you will see. None of that escape character bullshit either. If you enter `'\n'` you will see `"\\n"` in your program since it preserve the literal value!

```
echo -E '"\hello!\n"'


# In the program the argument will be ["\"\\hello!\\n\""]
```

However, the downside of using this is that you cannot put a single quote literal in between. Even if you use backslash! It will not work.

### Workaround with ' using ANSI-C Quoting

If you want to use single quotes in the parameter you are passing then you might want to use the `$''` ANSI-C quoting for parameter passing.

With `$''` you can pass a string with single quotes in them like so:

```
echo -E $'\'helloworld\\n'
```

However, then you will lose the literal preserving property of single quotes and then you will have to escape single quotes, backslashes and other special characters, accordingly because they will be treated as special characters if escaped!

**You cannot do variable interpolation in ANSI-C Quoting! If you want to do variable interpolation use double quotes.**

## Double Quote

Using double quote it will preserve the literal value as well, except $, `, \ which will expand those variables or commands.

Backslashes will escape the following special characters and will be removed: *$*, ", ` *(backtik)*, \, *<newline>*

```
echo -E "\$\`$\""

# In the program the argument is ["$`\""] because dollar sign, backtik don't
# have special meaning in the program. However, the quotes are escaped
# because they have special meaning. Note this is more for context of
# ruby programming when you are printing the literal string.
```

**Otherwise, if you backslash a non special character, the backslash will stay there as well as the non special character.**

```
./myProgram "\n" vs ./myProgram "\\n"

# The argument passed in both cases are the same! ["\\n"] because again since
# the backslash is preceded by n which is not one of the special characters
# that double quotes knows how to escape, it will be treated as string literals!
# For the first case. For the second case because you escaped the escape character
# it will be treated as string literals, so they have the same effect in the end!
# surprise pikachu.
```

# Aside: Passing \n as Bash Argument

To pass the actual newline character not the literal in Bash you can do it in the following ways.

```
./myProgram "
"


OR


./myProgram $'\n'
```

Do not try to pass it via `./myProgram "\\n"` this will escape the backslash so your actual string that gets passed is "\\n", a length of 2!

# Aside: zsh echo

The zsh version of the echo will automatically append the `-e` flag which enable interpretation of the escape characters. Which mean the escaped character will be interpreted again after you pass it in! Which often conflict with the examples provided in this chapter. So to prevent it either use `bash` or append the `-E` flag to prevent interpretation of the escape character again.

# Backslash Escapes

> See link for valid information. This page is outdated.

When you are working with special character arguments for say Python or Bash scripts, remember that if you only put one backslash it will escape the character that comes after it. In the case that the character that comes after it doesn't have any special meaning it will just interpret it as a character literal.

`./script.sh \q`, then *$1* will be the string literal "q" because there is no special escape character with \q.

If you want to pass a backslash, keep in that mind in order to pass the backslash literal to a program you have to type backslash twice to escape the escape character. So `\\` -> for one backslash literal interpretation.

In the programming language when you actually print the string it will only be one backslash printed, and the length of that string is only 1!

## Exception

The only exception I know so far is \n, if you want to pass \n in command line argument see the following post

Passing \n (not literal)

Otherwise, if you want to pass the string literal "\n" then you would have to pass \\n, the two backslash is interpreted as \ literal.

# find glob expansion

When using regular expression in find, you have to quote the name parameter to prevent glob expansion (the name expansion done by the shell)

```
find . -name *.java


vs


find . name "*.java"
```

The first method will expand to the files you have at the current directory, so say if you have a file named "hello.java", then the command you wrote will be expanded into `find . -name hello.java` which will only find "hello.java".

The second method when you quote the pattern, you prevent it from being expanded in the shell, and will correctly find all of the files that have .java extension.

# Echo Clear Previous Line

The escape string `\033[0K` will clear the entire line, this string are cross-platform so it will work on *zsh, bash, ...etc.*

For *zsh* you can use the string `\e[0K` instead of `033`

# Diff Command Line Tool

https://www.computerhope.com/unix/udiff.htm#How%20diff%20Works

# grep, awk, sed family tool

## Grep

Global Regular Expression Pattern

With grep you can do simple text-based or regular expression search on the file you passed or can be also piped.

**You can only provide in one pattern, you can provide in multiple pattern to search for by using the pipe symbol, but any parameter after the pattern are treated as filenames**

Use the `-E` flag to use the extended regular expression notation, i.e. you don't have to escape the following character to get their special meaning, *(), ?, +, {},* and | in the regular expression.

^ and $ anchor symbol to force the match in the beginning of the line and end of the line respectively. They do not need to be escaped.

In addition, if you escape non-special characters like `\a, \t` it will just be treated as literal of `a, t` respectively.

### Usages

| | |
|---|---|
| `grep <insert word to search> filename` | This will perform a search on the entire file to find where the word occurs |
| `grep -i <word> filename` | Perform a case-insensitive search on the entire file |
| `grep -R <word> .` | Perform a search on all of the file in the current directory as well as the sub-directories |
| `grep -c <word> filename` | Count the number of matches |
| `grep -A -B -C <word> filename` | Use -A, -B, and -C to get context surrounding the matched text, after, before, and both before and after respectively |

## Awk

The *awk* tool allows you to basically split each of line of your given text into different fields, kind of like *pandas* in Python. You can then access each columns individually allowing you to perform some level of numerical analysis.

The way to invoke *awk* is via:

`awk -F <delimiters> -f <awk program file/provide program in a single quoted string> -v var=4 <file to process>`

# Basic Structure of Awk

```
BEGIN {
# Applied before processing every line
}

/regex/ {
# Applied to only lines that match the regex
}

$1 ~ /regex/ {
# Applied to only lines that has it's first field match the regex
}

$3 == "hello" {
# Applied to only lines that has it's third field equal to hello
}

{
# Applied to every line
}

END {
# Applied after every line is ran
}
```

First the way that awk program is ran is through these pattern block:

The *BEGIN* and *END* pattern block are special in that they are only ran before processing through rest of the line and after every line is ran. If you want to do some setup work before processing through the rest of the line, i.e. setting up variables, you can do it in those two pattern block.

Then you can have a regex match based pattern block so that the code inside will only run if the lines matched the regex.

You can have nothing as well, so that the code is run for every line.

If you have multiple pattern block matched, they will all run, it is not like if-statement where one is true the rest are skipped, if they are all true, they are all run, from the order they are prescribed.

## Array In Awk

If you want to use an array in awk, just use it you don't need to declare it to use. Just start insert elements into the variable you want to use as an array.

# Sed

The *sed* tool or stream editor allows you to do textual replacement for the lines of files you have pass it.

The syntax for *sed* is kinda meh, but let's go over the different kind of operation you can do with it.

Each *sed* command follows the syntax:

`[addr]X[options]`

Where *X* is a single-letter *sed* command. *[addr]* is an optional line addressing, telling from where to where to apply the command.

- Address can be a single line number (to only execute on that line)
- A regular expression (to only execute on matched lines)
- Range of lines (to execute command between the two ranges)

Without *[addr]* the command is implicitly global, applying to every line.

## Two Common Sed Operations

1. **d (delete)**
   The delete operation will delete the lines specified. Again if no addresses then it is implicitly deleting every line
2. **s/regexp/replacement/flags**
   The substitute command allow you to specify the text or a regular expression that's matched to be replaced by whatever you want. You can also use groups within the regexp, but they have to be escaped otherwise, they will be interpreted literally.

   In addition, the *g* flag can be provided as the last argument to do every replacement possible across the entire line, otherwise, it will only replaced the first match it found.

## Addressing

There are different ways to do addressing, picking which line you are going to apply the command over.

1. Empty: If none, the command is global by default, meaning it applies to every line
2. One number: By providing one number you are only executing the command for that particular line
3. /regex/: If you provide a regular expression as the address, then it will only apply to the lines that matches the regular expression
4. start,end: You can also specify an address range using comma, the range can be numeric, regular expression, or even a mix of both.

   Ex: '4,17d': Delete line 4 to 17 inclusive.

Note: You can also append "!" to negate the address specification if you provide it right after the address. To pick every line that is not part of the provided address.

# Examples

```
# This command replaces cat with dog, however, it is only the first occurrence it finds.
sed 's/cat/dog/' input

# Same as previous, but replaces every occurrence of cat with dog in a line.
sed 's/cat/dog/g' input

# Only apply the substitution to lines that contains apple, and only replaces the first
occurence, from apply -> app.
sed '/apple/s/pple/pp/' input

# Delete line 3-4 inclusive
sed '3,4d' input

# Delete every line
sed 'd' input
```

# Command substitution vs Process substitution

https://unix.stackexchange.com/a/393352

# tr and cut command

## tr

The `tr` command is used to translate or delete characters that comes from the standard input.

If say you would like to replace every character in a file with another character then you can just do something like:

```
echo "hello world" | tr e z // Prints out "hzllo world"
```

If you would like to delete characters then you would use the `-d` flag and provide in the set of characters that you would like to delete like such:

```
echo "hello world" | tr -d e // Prints out hllo world
```

## cut

The cut command is used like the split function. To extract out the field that you want from a particular file or from standard input. For example:

```
echo "hello-world-lol" | cut -d "-" -f1
```

This will split the string according to the delimiter "-", then output the first field after splitting which is just "hello". You must specify the field that you would like to extract after splitting.

# Quoting and Not Quoting Arguments

## Word splitting

Bash shell and zsh shell will scan results of

1. Parameter expansion: `$PATH`
2. Command substitution: `$(echo hi)`
3. Arithmetic expansion: `$((x += 1))`

For word splitting if they DO NOT occur within double quotes.

Basically with word splitting the shell will treat whatever is in `$IFS` as delimiter for word splitting, it is usually spaces, newline, and tab. It will split those results into different arguments for example "one two three" will become three parameter and be passed into whatever the program it is invoked with. For example

```
args="one two three"
./myprogram $args
```

`myprogram` will be passed a total of THREE arguments, "one", "two", and "three" respectively. However, if you do `./myprogram "$args"` it will instead be passed with only one argument.

# How does Program Execute in Linux

The shell forks itself then exec to replace the child with the program that we would like to run. The parent which is the shell, calls wait on the child in order to wait for it's return value. The stdin, stdout are also connected in order for the outputs from the child to be displayed properly. Otherwise, you wouldn't be able to see the output of find command. For example.

https://unix.stackexchange.com/questions/225736/how-does-a-shell-execute-a-program

https://stackoverflow.com/questions/4204915/please-explain-the-exec-function-and-its-family

https://fsl.fmrib.ox.ac.uk/fslcourse/unix_intro/shell.html -> very nice simple explanation of how unix interpreter works. Interpreter can be used interchangeably with shell. They are referring to the same thing, a program that reads in prompt, execute the program, wait for child, and repeat.

https://www.reddit.com/r/bash/comments/ugoz97/today_i_understood_the_importance_of_the_shebang/ -> Explains what is shebang