Table of Contents

Linux Command Line (Bash)

Bash is a name of the Unix shell, which was also distributed as the shell for the GNU operating system and as the default shell on most Linux distros. Nearly all examples below can be a part of a shell script or executed directly in the shell.

When you open terminal you see something like

- `user_name@computer_name:~$` this what you will see most of the time.

If you want to login as a **root** (superuser or administrator) you type `sudo -s` (or `su -`) then enter root password (if of course you have it) and then you will see

- `root@computer_name:~#`

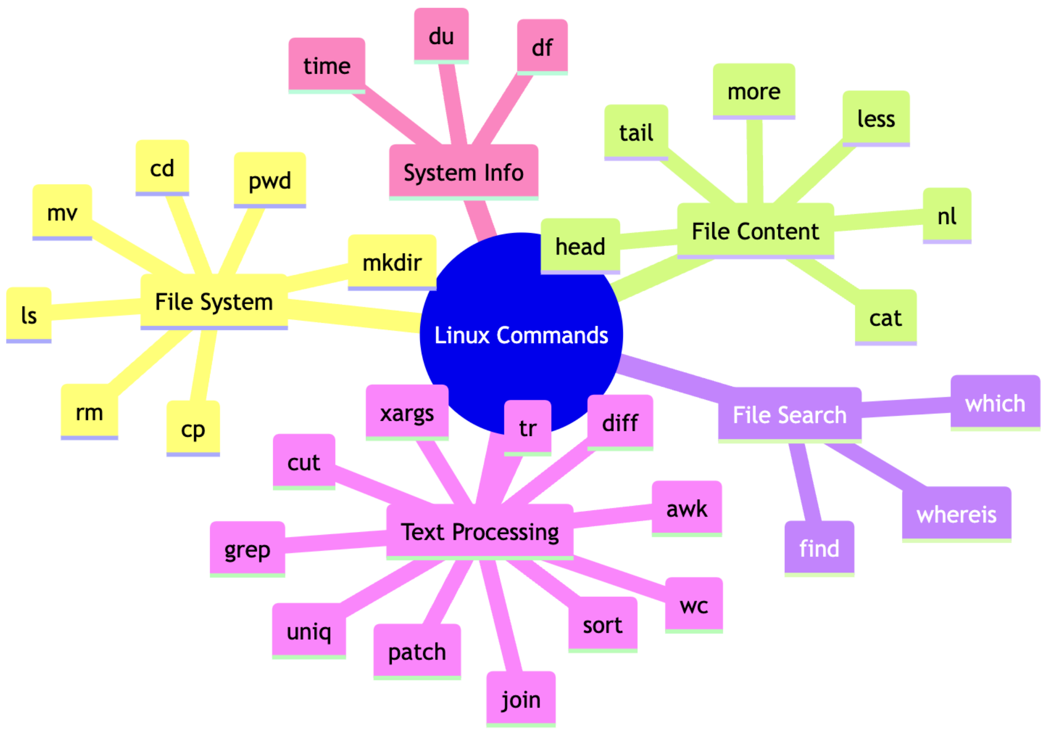
As you can see `$` is the sign of being a regular user and `#` is the sign of being a root/superuser. The sign `~` (tilda) is the path of the home directory. That means that `~ = /home/user_name/`

Linux is a multi-user and time-sharing OS, therefore, several users can work on the same machine simultaneously. If you have logged-in as a certain user with e.g. `user_name1`, if you want to change the user and login as another user, you use `sudo` command. Suppose you have opened the terminal in

- `user_name1@computer_name:~$ sudo user_name2`, then put the **login password** of user `user_name2`
- it will return `user_name2@computer_name:~$`

Commands

- `whoami` - prints the name of the currently logged-in user in the terminal session.
- `man` - prints the manual or information about a command (e.g. `$man whoami`).
- `clear` - clears the screen from previous commands in the terminal.
- `pwd` - shows the path of the present working directory (**P**resent **W**orking **D**irectory).
- `cd` - changes directory to a given path (**C**urrent **D**irectory).
- `ls` - list content of the current directory (options `ls -a` all files, `ls -l` shows file permissions).
- `mkdir` - make a directory
- `rmdir` - remove directory
- `rm` - remove files (`rm -r` removes recursively, `rm -v` gives feedback, `rm -i` requires confirmation to delete files and prevents from accidentally deleting files, `rm -rf` removes recursively with force, it can delete anything given that you have permissions).
- `cp` - copy file from one directory to another (`cp -r` copy recursively)
- `touch` - change file timestamps. It can be also used to create an empty file.
- `mv` - move or rename files/directories
- `head` - prints the first 10 lines (header) of a file (e.g. `head -n 20` shows first 20 lines)
- `tail` - prints the last 10 lines of a file (e.g. `tail -n 20` shows last 20 lines). If file is appended continuously with new lines at the end you can use `tail -f` to print in real time the new lines.
- `cat` - can add content to a file, or by default can show the content of a file.
- `less` - shows file content and you can scroll it bit by bit. Use **b** to scroll one page, **G** to go to the end, **g** to go to the start and **q** to quit.
- `more` - shows all file content.
- `echo` - prints the argument passed to it.
- `wc` - stands for word count. `wc -l` prints newline count. `wc -m` prints character count. `wc -c` prints bytes, `wc -w` prints word count.
- `grep` - one of the most widely used text manipulation command. It lets you filter the content of a file for display (`$cat file | grep word`).
- `>` - redirects standard output, allows to send the commands output to e.g. a file `$whoami > file`.
- `>>` - redirects standard outputs and appends new contents. This allow that the new content appends e.g. at the end of the file `$whoami > file`, `$pwd >> file` add in another line.
- `|` - this operator is called pipe. This takes the output of one command and passes it as the input for
  another command (`cat file | grep output`).



Ref.: labex.io

The following cheat sheet is taken from https://www.geeksforgeeks.org/linux-commands-cheat-sheet/

Common File and Directory Operations Commands

File and directory operations are fundamental in working with the Linux operating system. Here are some commonly used File and Directory Operations commands.

> **Note**
>
> When changing directories the
>
> - `.` is the current directory path.
> - `..` is the path of a directory above in the tree level (mother directory).
> - `~` is the path of the home directory (`/home/user_name/`).
> - `/` is the root path of all trees.

Command	Description	Options	Examples
ls	List files and directories.	• **-l**: Long format listing. • **-a**: Include hidden files hidden ones • **-h**: Human-readable file sizes.	• **ls -l** displays files and directories with detailed information. • **ls -a** shows all files and directories, including • **ls -lh** displays file sizes in a human-readable format.
cd	Change directory.		• **cd /path/to/directory** changes the current directory to the specified path.
pwd	Print current working directory.		• **pwd** displays the current working directory.
mkdir	Create a new directory.		• **mkdir my_directory** creates a new directory named “my_directory”.
rm	Remove files and directories.	• **-r**: Remove directories recursively. • **-f**: Force removal without confirmation.	• **rm file.txt** deletes the file named “file.txt”. • **rm -r my_directory** deletes the directory “my_directory” and its contents. • **rm -f file.txt** forcefully deletes the file “file.txt” without confirmation.
cp	Copy files and directories.	• **-r**: Copy directories recursively.	• **cp -r directory destination** copies the directory “directory” and its contents to the specified destination. • **cp file.txt destination** copies the file “file.txt” to the specified destination.

Command	Description	Options	Examples
mv	Move/rename files and directories.		• **mv file.txt new_name.txt** renames the file “file.txt” to “new_name.txt”. • **mv file.txt directory** moves the file “file.txt” to the specified directory.
touch	Create an empty file or update file timestamps.		• **touch file.txt** creates an empty file named “file.txt”.
cat	View the contents of a file.		• **cat file.txt** displays the contents of the file “file.txt”.
head	Display the first few lines of a file.	• **-n**: Specify the number of lines to display.	• **head file.txt** shows the first 10 lines of the file “file.txt”. • **head -n 5 file.txt** displays the first 5 lines of the file “file.txt”.
tail	Display the last few lines of a file.	• **-n**: Specify the number of lines to display.	• **tail file.txt** shows the last 10 lines of the file “file.txt”. • **tail -n 5 file.txt** displays the last 5 lines of the file “file.txt”.
ln	Create links between files.	• **-s**: Create symbolic (soft) links.	• **ln -s source_file link_name** creates a symbolic link named “link_name” pointing to “source_file”.
find	Search for files and directories.	• **-name**: Search by filename. • **-type**: Search by file type.	• **find /path/to/search -name “*.txt”** searches for all files with the extension “.txt” in the specified directory.

File Permission Commands

File permissions on Linux and Unix systems control access to files and directories. There are three basic permissions: read, write, and execute. Each permission can be granted or denied to three different categories of users: the owner of the file, the members of the file’s group, and everyone else.

Here are some file permission commands:

Command	Description	Options	Examples
chmod	Change file permissions.	• **u**: User/owner permissions. • **g**: Group permissions. • **o**: Other permissions. • **+**: Add permissions. • **–**: Remove permissions. • **=**: Set permissions explicitly.	• **chmod u+rwx file.txt** grants read, write, and execute permissions to the owner of the file.
chown	Change file ownership.		• **chown user file.txt** changes the owner of "file.txt" to the specified user.
chgrp	Change group ownership.		• **chgrp group file.txt** changes the group ownership of "file.txt" to the specified group.
umask	Set default file permissions.		• **umask 022** sets the default file permissions to read and write for the owner, and read-only for group and others.

Compression and Archiving Commands

Here are some file compression and archiving commands in Linux:

Commands	Description	Options	Examples
tar	Create or extract archive files.	• **-c**: Create a new archive. • **-x**: Extract files from an archive. • **-f**: Specify the archive file name. • **-v**: Verbose mode. • **-z**: Compress the archive with gzip. • **-j**: Compress the archive with bzip2.	• **tar -czvf archive.tar.gz files/** creates a compressed tar archive named “archive.tar.gz” containing the files in the “files/” directory.
gzip	Compress files.	• **-d**: Decompress files.	• **gzip file.txt** compresses the file “file.txt” and renames it as “file.txt.gz”.
zip	Create compressed zip archives.	• **-r**: Recursively include directories.	• **zip archive.zip file1.txt file2.txt** creates a zip archive named “archive.zip” containing “file1.txt” and “file2.txt”.

Process Management Commands

In Linux, process management commands allow you to monitor and control running processes on the system. Here are some commonly used process management commands:

Commands	Description	Options	Examples
ps	Display running processes.	• **-aux**: Show all processes.	• **ps aux** shows all running processes with detailed information.

Commands	Description	Options	Examples
top	Monitor system processes in real-time.		• **top** displays a dynamic view of system processes and their resource usage.
kill	Terminate a process.	• **-9**: Forcefully kill a process.	• **kill PID** terminates the process with the specified process ID.
pkill	Terminate processes based on their name.		• **pkill process_name** terminates all processes with the specified name.
pgrep	List processes based on their name.		• **pgrep process_name** lists all processes with the specified name.

Commands	Description	Options	Examples
grep	used to search for specific patterns or regular expressions in text files or streams and display matching lines.	• **-i**: Ignore case distinctions while searching. • **-v**: Invert the match, displaying non-matching lines. • **-r or -R**: Recursively search directories for matching patterns. • **-l**: Print only the names of files containing matches. • **-n**: Display line numbers alongside matching lines. • **-w**: Match whole words only, rather than partial matches. • **-c**: Count the number of matching lines instead of displaying them. • **-e**: Specify multiple patterns to search for. • **-A**: Display lines after the matching line. • **-B**: Display lines before the matching line. • **-C**: Display lines both before and after the matching line.	• **grep -i "hello" file.txt** • **grep -v "error" file.txt** • **grep -r "pattern" directory/** • **grep -l "keyword" file.txt** • **grep -n "pattern" file.txt** In these examples we are extracting our desirec output from filename (file.txt)

System Information Commands

In Linux, there are several commands available to gather system information. Here are some commonly used system information commands:

sudCommand	Description	Options	Examples
uname	Print system information.	• **-a**: All system information.	• **uname -a** displays all system information.
whoami	Display current username.		• **whoami** shows the current username.
df	Show disk space usage.	• **-h**: Human-readable sizes.	• **df -h** displays disk space usage in a human-readable format.
du	Estimate file and directory sizes.	• **-h**: Human-readable sizes. • **-s**: Display total size only.	• **du -sh directory/** provides the total size of the specified directory.
free	Display memory usage information.	• **-h**: Human-readable sizes.	• **free -h** displays memory usage in a human-readable format.
uptime	Show system uptime.		• **uptime** shows the current system uptime.
lscpu	Display CPU information.		• **lscpu** provides detailed CPU information.
lspci	List PCI devices.		• **lspci** List PCI devices.
lsusb	List USB devices.		• **lsusb** lists all connected USB devices.

Networking Commands

In Linux, there are several networking commands available to manage and troubleshoot network connections. Here are some commonly used networking commands:

Command	Description	Examples
ifconfig	Display network interface information.	• **ifconfig** shows the details of all network interfaces.
ping	Send ICMP echo requests to a host.	• **ping google.com** sends ICMP echo requests to “google.com” to check connectivity.
netstat	Display network connections and statistics.	• **netstat -tuln** shows all listening TCP and UDP connections.
ss	Display network socket information.	• **ss -tuln** shows all listening TCP and UDP connections.
ssh	Securely connect to a remote server.	• **ssh user@hostname** initiates an SSH connection to the specified hostname.
scp	Securely copy files between hosts.	• **scp file.txt user@hostname:/path/to/destination** securely copies “file.txt” to the specified remote host.
wget	Download files from the web.	• **wget http://example.com/file.txt** downloads “file.txt” from the specified URL.
curl	Transfer data to or from a server.	• **curl http://example.com** retrieves the content of a webpage from the specified URL.

I/O Redirection Commands

In Linux, IO (Input/Output) redirection commands are used to redirect the standard input, output, and error streams of commands and processes. Here are some commonly used IO redirection commands:

Command	Description
cmd < file	Input of cmd is taken from file.
cmd > file	Standard output (stdout) of cmd is redirected to file.

Command	Description
cmd 2> file	Error output (stderr) of cmd is redirected to file.
cmd 2>&1	stderr is redirected to the same place as stdout.
cmd1 <(cmd2)	Output of cmd2 is used as the input file for cmd1.
cmd > /dev/null	Discards the stdout of cmd by sending it to the null device.
cmd &> file	Every output of cmd is redirected to file.
cmd 1>&2	stdout is redirected to the same place as stderr.
cmd >> file	Appends the stdout of cmd to file.

Environment Variable Commands

In Linux, environment variables are used to store configuration settings, system information, and other variables that can be accessed by processes and shell scripts. Here are some commonly used environment variable commands:

Command	Description
export VARIABLE_NAME=value	Sets the value of an environment variable.
echo $VARIABLE_NAME	Displays the value of a specific environment variable.
env	Lists all environment variables currently set in the system.
unset VARIABLE_NAME	Unsets or removes an environment variable.
export -p	Shows a list of all currently exported environment variables.
env VAR1=value COMMAND	Sets the value of an environment variable for a specific command.
printenv	Displays the values of all environment variables.

User Management Commands

In Linux, user management commands allow you to create, modify, and manage user accounts on the system. Here are some commonly used user management commands:

Command	Description
who	Show who is currently logged in.

Command	Description
sudo adduser username	Create a new user account on the system with the specified username.
finger	Display information about all the users currently logged into the system, including their usernames, login time, and terminal.
sudo deluser USER GROUPNAME	Remove the specified user from the specified group.
last	Show the recent login history of users.
finger username	Provide information about the specified user, including their username, real name, terminal, idle time, and login time.
sudo userdel -r username	Delete the specified user account from the system, including their home directory and associated files. The -r option ensures the removal of the user's files.
sudo passwd -l username	Lock the password of the specified user account, preventing the user from logging in.
su – username	Switch to another user account with the user's environment.
sudo usermod -a -G GROUPNAME USERNAME	Add an existing user to the specified group. The user is added to the group without removing them from their current groups.

Bash Shortcuts Commands

There are many shortcuts commands in Linux that can help you be more productive. Here are a few of the most common ones:

Navigation	Description	Editing	Description	History	Description
Ctrl + A	Move to the beginning of the line.	**Ctrl + U**	Cut/delete from the cursor position to the beginning of the line.	**Ctrl + R**	Search command history (reverse search).
Ctrl + E	Move to the end of the line.	**Ctrl + K**	Cut/delete from the cursor position to the end of the line.	**Ctrl + G**	Escape from history search mode.
Ctrl + B	Move back one character.	**Ctrl + W**	Cut/delete the word before the cursor.	**Ctrl + P**	Go to the previous command in history.

Navigation	Description	Editing	Description	History	Description
Ctrl + F	Move forward one character.	**Ctrl + Y**	Paste the last cut text.	**Ctrl + N**	Go to the next command in history.
Alt + B	Move back one word	**Ctrl + L**	Clear the screen.	**Ctrl + C**	Terminate the current command.
Alt + F	Move forward one word.				

Exercises

Source: https://itp.uni-frankfurt.de/~gros/Vorlesungen/CPP/sheet1.pdf

1. **Explore Filesystem:**
   - Use the commands `cd`, `pwd` and `ls` to explore the filesystem.
   - Go to the `/etc` directory and see what is there, check the rest of the filesystem tree using `cd`, `ls`, `pwd` and `cat`. Look in `/bin`, `/usr/bin`, `/sbin`, `/tmp` and `/boot`.
   - Go to your home directory and generate a directory called `uni` and `notuni`, with the `mkdir` command. Change to `notuni` and generate a file with `touch newfile`.
   - Copy the file `newfile` to `copyofnewfile` in the directory `notuni` using the command `cp`. Then rename the file by moving the file with the command `mv`.
   - Now go back one level try to delete both directories using `rm` and its options (check `man rm`).
   - What is the difference between listing the contents with `ls -ltr` and `ls -l`, or `ls` (check some of the options, often denoted flags listed in `man ls`)
2. **Reading from files:** Navigate to a directory that contains a text file
   - Show the content of the text file one page at a time using the `less <filename>` command.
   - Show the first and last ten lines of the file using `head <filename>` and `tail <filename>`, respectively.
   - Use `grep <word> <filename>` to search your file for lines containing a word of your choice.
3. **Permissions**
   - Create a file and a directory with permissions `r--r--r--`. Can you change to the directory you created now? (hint: `man chmod`).

    - *Modify the permissions on your home directory to make it completely private. Check with some other user that he can't access your directory. Then put the permissions back to how they were. Choose a directory in your home directory and make all the files on it read only.*
4. **Connecting to other computers**
    - Log into another machine in the lab. Find out the name of your lab computer (use `hostname` and `domainname`). Go to another machine and then log to the first machine using `ssh`: (`ssh username@machine.domain`). You can then work in the other computer.
    - Save one step and run a command in the remote computer via `ssh username@machine.domain <command>`. See what happens if you end the connection while the program is running. Try with a graphical program, for instance `Firefox` (you need an extra option for that, see `man ssh`).
5. **Pipe operator and `xargs`** The pipe operator `|` is used to hand over the output of one command to
another command. Navigate to a directory of your choice.
    - Use the `grep` command and pipe operator to filter the list of files by a specific keyword. For example, if you wanted to filter by files containing some `<word>`, you would use the command `ls | grep <word>`.
    - Pipe the output of `ls -lah` into the `less` command.
    - If the output of a command contains different arguments that you wish to pipe into another command, the `xargs` utility can be useful. Use `printf "test1.txt\ntest2.txt"` to print out two file names separated by a newline (`\n`).
    - Now pipe the output of that command into `xargs` using the following command: `printf "test1.txt\ntest2.txt" | xargs touch`. The `xargs` utility will, for each line of the output, execute a given command (here `touch`) with that line as an argument. If everything worked, two new files `test1.txt` and `test2.txt` were created. Check if that is the case using `ls`.

Read more here.

```
#!/usr/bin/env bash
# First line of the script is the shebang which tells the system how to execute
# the script: https://en.wikipedia.org/wiki/Shebang_(Unix)
# As you already figured, comments start with #. Shebang is also a comment.

# Simple hello world example:
echo "Hello world!" # => Hello world!

# Each command starts on a new line, or after a semicolon:
echo "This is the first command"; echo "This is the second command"
# => This is the first command
# => This is the second command
```

```bash
# Declaring a variable looks like this:
variable="Some string"

# But not like this:
variable = "Some string" # => returns error "variable: command not found"
# Bash will decide that `variable` is a command it must execute and give an error
# because it can't be found.

# Nor like this:
variable= "Some string" # => returns error: "Some string: command not found"
# Bash will decide that "Some string" is a command it must execute and give an
# error because it can't be found. In this case the "variable=" part is seen
# as a variable assignment valid only for the scope of the "Some string"
# command.

# Using the variable:
echo "$variable" # => Some string
echo '$variable' # => $variable
# When you use a variable itself — assign it, export it, or else — you write
# its name without $. If you want to use the variable's value, you should use $.
# Note that ' (single quote) won't expand the variables!
# You can write variable without surrounding quotes but it's not recommended.

# Parameter expansion ${...}:
echo "${variable}" # => Some string
# This is a simple usage of parameter expansion such as two examples above.
# Parameter expansion gets a value from a variable.
# It "expands" or prints the value.
# During the expansion time the value or parameter can be modified.
# Below are other modifications that add onto this expansion.

# String substitution in variables:
echo "${variable/Some/A}" # => A string
# This will substitute the first occurrence of "Some" with "A".

# Substring from a variable:
length=7
echo "${variable:0:length}" # => Some st
# This will return only the first 7 characters of the value
echo "${variable: -5}" # => tring
# This will return the last 5 characters (note the space before -5).
# The space before minus is mandatory here.

# String length:
echo "${#variable}" # => 11

# Indirect expansion:
other_variable="variable"
echo ${!other_variable} # => Some string
# This will expand the value of `other_variable`.

# The default value for variable:
echo "${foo:-"DefaultValueIfFooIsMissingOrEmpty"}"
# => DefaultValueIfFooIsMissingOrEmpty
# This works for null (foo=) and empty string (foo=""); zero (foo=0) returns 0.
# Note that it only returns default value and doesn't change variable value.
# Declare an array with 6 elements:
array=(one two three four five six)
# Print the first element:
echo "${array[0]}" # => "one"
# Print all elements:
echo "${array[@]}" # => "one two three four five six"
# Print the number of elements:
echo "${#array[@]}" # => "6"
# Print the number of characters in third element
echo "${#array[2]}" # => "5"
# Print 2 elements starting from fourth:
echo "${array[@]:3:2}" # => "four five"
# Print all elements each of them on new line.
for item in "${array[@]}"; do
    echo "$item"
done

# Built-in variables:
# There are some useful built-in variables, like:
echo "Last program's return value: $?"
echo "Script's PID: $$"
echo "Number of arguments passed to script: $#"
echo "All arguments passed to script: $@"
echo "Script's arguments separated into different variables: $1 $2..."

# Brace Expansion {...}
# used to generate arbitrary strings:
echo {1..10} # => 1 2 3 4 5 6 7 8 9 10
echo {a..z} # => a b c d e f g h i j k l m n o p q r s t u v w x y z
# This will output the range from the start value to the end value.
# Note that you can't use variables here:
from=1
to=10
echo {$from..$to} # => {$from..$to}

# Now that we know how to echo and use variables,
# let's learn some of the other basics of Bash!

# Our current directory is available through the command `pwd`.
# `pwd` stands for "print working directory".
# We can also use the built-in variable `$PWD`.
# Observe that the following are equivalent:
echo "I'm in $(pwd)" # execs `pwd` and interpolates output
echo "I'm in $PWD" # interpolates the variable

# If you get too much output in your terminal, or from a script, the command
# `clear` clears your screen:
clear
# Ctrl-L also works for clearing output.

# Reading a value from input:
echo "What's your name?"
read name
# Note that we didn't need to declare a new variable.
echo "Hello, $name!"
```

```
# We have the usual if structure.
# Condition is true if the value of $name is not equal to the current user's
login username:
if [[ "$name" != "$USER" ]]; then
    echo "Your name isn't your username"
else
    echo "Your name is your username"
fi

# To use && and || with if statements, you need multiple pairs of square
brackets:
read age
if [[ "$name" == "Steve" ]] && [[ "$age" -eq 15 ]]; then
    echo "This will run if $name is Steve AND $age is 15."
fi

if [[ "$name" == "Daniya" ]] || [[ "$name" == "Zach" ]]; then
    echo "This will run if $name is Daniya OR Zach."
fi
# There are other comparison operators for numbers listed below:
# -ne - not equal
# -lt - less than
# -gt - greater than
# -le - less than or equal to
# -ge - greater than or equal to

# There is also the `=~` operator, which tests a string against the Regex
pattern:
email=me@example.com
if [[ "$email" =~ [a-z]+@[a-z]{2,}\.(com|net|org) ]]
then
    echo "Valid email!"
fi

# There is also conditional execution
echo "Always executed" || echo "Only executed if first command fails"
# => Always executed
echo "Always executed" && echo "Only executed if first command does NOT fail"
# => Always executed
# => Only executed if first command does NOT fail

# A single ampersand & after a command runs it in the background. A background
command's
# output is printed to the terminal, but it cannot read from the input.
sleep 30 &
# List background jobs
jobs # => [1]+  Running                 sleep 30 &
# Bring the background job to the foreground
fg
# Ctrl-C to kill the process, or Ctrl-Z to pause it
# Resume a background process after it has been paused with Ctrl-Z
bg
# Kill job number 2
kill %2
# %1, %2, etc. can be used for fg and bg as well
# Redefine command `ping` as alias to send only 5 packets
alias ping='ping -c 5'
# Escape the alias and use command with this name instead
\ping 192.168.1.1
# Print all aliases
alias -p

# Expressions are denoted with the following format:
echo $(( 10 + 5 )) # => 15

# Unlike other programming languages, bash is a shell so it works in the context
# of a current directory. You can list files and directories in the current
# directory with the ls command:
ls # Lists the files and subdirectories contained in the current directory

# This command has options that control its execution:
ls -l # Lists every file and directory on a separate line
ls -t # Sorts the directory contents by last-modified date (descending)
ls -R # Recursively `ls` this directory and all of its subdirectories

# Results (stdout) of the previous command can be passed as input (stdin) to the
next command
# using a pipe |. Commands chained in this way are called a "pipeline", and are
run concurrently.
# The `grep` command filters the input with provided patterns.
# That's how we can list .txt files in the current directory:
ls -l | grep "\.txt"

# Use `cat` to print files to stdout:
cat file.txt

# We can also read the file using `cat`:
Contents=$(cat file.txt)
# "\n" prints a new line character
# "-e" to interpret the newline escape characters as escape characters
echo -e "START OF FILE\n$Contents\nEND OF FILE"
# => START OF FILE
# => [contents of file.txt]
# => END OF FILE

# Use `cp` to copy files or directories from one place to another.
# `cp` creates NEW versions of the sources,
# so editing the copy won't affect the original (and vice versa).
# Note that it will overwrite the destination if it already exists.
cp srcFile.txt clone.txt
cp -r srcDirectory/ dst/ # recursively copy

# Look into `scp` or `sftp` if you plan on exchanging files between computers.
# `scp` behaves very similarly to `cp`.
# `sftp` is more interactive.

# Use `mv` to move files or directories from one place to another.
# `mv` is similar to `cp`, but it deletes the source.
# `mv` is also useful for renaming files!
mv s0urc3.txt dst.txt # sorry, l33t hackers...
```

```bash
# Since bash works in the context of a current directory, you might want to
# run your command in some other directory. We have cd for changing location:
cd ~    # change to home directory
cd      # also goes to home directory
cd ..   # go up one directory
        # (^^say, from /home/username/Downloads to /home/username)
cd /home/username/Documents   # change to specified directory
cd ~/Documents/..    # now in home directory (if ~/Documents exists)
cd -    # change to last directory
# => /home/username/Documents

# Use subshells to work across directories
(echo "First, I'm here: $PWD") && (cd someDir; echo "Then, I'm here: $PWD")
pwd # still in first directory

# Use `mkdir` to create new directories.
mkdir myNewDir
# The `-p` flag causes new intermediate directories to be created as necessary.
mkdir -p myNewDir/with/intermediate/directories
# if the intermediate directories didn't already exist, running the above
# command without the `-p` flag would return an error

# You can redirect command input and output (stdin, stdout, and stderr)
# using "redirection operators". Unlike a pipe, which passes output to a command,
# a redirection operator has a command's input come from a file or stream, or
# sends its output to a file or stream.

# Read from stdin until ^EOF$ and overwrite hello.py with the lines
# between "EOF" (which are called a "here document"):
cat > hello.py << EOF
#!/usr/bin/env python
from __future__ import print_function
import sys
print("#stdout", file=sys.stdout)
print("#stderr", file=sys.stderr)
for line in sys.stdin:
    print(line, file=sys.stdout)
EOF
# Variables will be expanded if the first "EOF" is not quoted

# Run the hello.py Python script with various stdin, stdout, and
# stderr redirections:
python hello.py < "input.in" # pass input.in as input to the script

python hello.py > "output.out" # redirect output from the script to output.out

python hello.py 2> "error.err" # redirect error output to error.err

python hello.py > "output-and-error.log" 2>&1
# redirect both output and errors to output-and-error.log
# &1 means file descriptor 1 (stdout), so 2>&1 redirects stderr (2) to the
current
# destination of stdout (1), which has been redirected to output-and-error.log.

python hello.py > /dev/null 2>&1
```

```bash
# redirect all output and errors to the black hole, /dev/null, i.e., no output

# The output error will overwrite the file if it exists,
# if you want to append instead, use ">>":
python hello.py >> "output.out" 2>> "error.err"

# Overwrite output.out, append to error.err, and count lines:
info bash 'Basic Shell Features' 'Redirections' > output.out 2>> error.err
wc -l output.out error.err

# Run a command and print its file descriptor (e.g. /dev/fd/123)
# see: man fd
echo <(echo "#helloworld")

# Overwrite output.out with "#helloworld":
cat > output.out <(echo "#helloworld")
echo "#helloworld" > output.out
echo "#helloworld" | cat > output.out
echo "#helloworld" | tee output.out >/dev/null

# Cleanup temporary files verbosely (add '-i' for interactive)
# WARNING: `rm` commands cannot be undone
rm -v output.out error.err output-and-error.log
rm -r tempDir/ # recursively delete
# You can install the `trash-cli` Python package to have `trash`
# which puts files in the system trash and doesn't delete them directly
# see https://pypi.org/project/trash-cli/ if you want to be careful

# Commands can be substituted within other commands using $( ):
# The following command displays the number of files and directories in the
# current directory.
echo "There are $(ls | wc -l) items here."

# The same can be done using backticks `` but they can't be nested -
# the preferred way is to use $( ).
echo "There are `ls | wc -l` items here."

# Bash uses a `case` statement that works similarly to switch in Java and C++:
case "$Variable" in
    # List patterns for the conditions you want to meet
    0) echo "There is a zero.";;
    1) echo "There is a one.";;
    *) echo "It is not null.";;  # match everything
esac

# `for` loops iterate for as many arguments given:
# The contents of $Variable is printed three times.
for Variable in {1..3}
do
    echo "$Variable"
done
# => 1
# => 2
# => 3
# Or write it the "traditional for loop" way:
for ((a=1; a <= 3; a++))
do
    echo $a
done
# => 1
# => 2
# => 3

# They can also be used to act on files..
# This will run the command `cat` on file1 and file2
for Variable in file1 file2
do
    cat "$Variable"
done

# ..or the output from a command
# This will `cat` the output from `ls`.
for Output in $(ls)
do
    cat "$Output"
done

# Bash can also accept patterns, like this to `cat`
# all the Markdown files in current directory
for Output in ./*.markdown
do
    cat "$Output"
done

# while loop:
while [ true ]
do
    echo "loop body here..."
    break
done
# => loop body here...

# You can also define functions
# Definition:
function foo ()
{
    echo "Arguments work just like script arguments: $@"
    echo "And: $1 $2..."
    echo "This is a function"
    returnValue=0    # Variable values can be returned
    return $returnValue
}
# Call the function `foo` with two arguments, arg1 and arg2:
foo arg1 arg2
# => Arguments work just like script arguments: arg1 arg2
# => And: arg1 arg2...
# => This is a function
# Return values can be obtained with $?
resultValue=$?
# More than 9 arguments are also possible by using braces, e.g. ${10}, ${11}, ...
# or simply
bar ()
{
    echo "Another way to declare functions!"
    return 0
}
# Call the function `bar` with no arguments:
bar # => Another way to declare functions!

# Calling your function
foo "My name is" $Name

# There are a lot of useful commands you should learn:
# prints last 10 lines of file.txt
tail -n 10 file.txt

# prints first 10 lines of file.txt
head -n 10 file.txt

# print file.txt's lines in sorted order
sort file.txt

# report or omit repeated lines, with -d it reports them
uniq -d file.txt

# prints only the first column before the ',' character
cut -d ',' -f 1 file.txt

# replaces every occurrence of 'okay' with 'great' in file.txt
# (regex compatible)
sed -i 's/okay/great/g' file.txt
# be aware that this -i flag means that file.txt will be changed
# -i or --in-place erase the input file (use --in-place=.backup to keep a back-
up)

# print to stdout all lines of file.txt which match some regex
# The example prints lines which begin with "foo" and end in "bar"
grep "^foo.*bar$" file.txt

# pass the option "-c" to instead print the number of lines matching the regex
grep -c "^foo.*bar$" file.txt

# Other useful options are:
grep -r "^foo.*bar$" someDir/ # recursively `grep`
grep -n "^foo.*bar$" file.txt # give line numbers
grep -rI "^foo.*bar$" someDir/ # recursively `grep`, but ignore binary files

# perform the same initial search, but filter out the lines containing "baz"
grep "^foo.*bar$" file.txt | grep -v "baz"

# if you literally want to search for the string,
# and not the regex, use `fgrep` (or `grep -F`)
fgrep "foobar" file.txt

# The `trap` command allows you to execute a command whenever your script
```

```bash
# receives a signal. Here, `trap` will execute `rm` if it receives any of the
# three listed signals.
trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM

# `sudo` is used to perform commands as the superuser
# usually it will ask interactively the password of superuser
NAME1=$(whoami)
NAME2=$(sudo whoami)
echo "Was $NAME1, then became more powerful $NAME2"

# Read Bash shell built-ins documentation with the bash `help` built-in:
help
help help
help for
help return
help source
help .

# Read Bash manpage documentation with `man`
apropos bash
man 1 bash
man bash

# Read info documentation with `info` (`?` for help)
apropos info | grep '^info.*('
man info
info info
info 5 info

# Read bash info documentation:
info bash
info bash 'Bash Features'
info bash 6
info --apropos bash
```