Linux Command Line

For Engineers and Researchers

Marcos Borges, PhD

Linux Command Line
For Engineers and Researchers

Your feedback matters

Thank you for reading this free book about the Linux Command Line.

It was created with great care and many hours of work to make learning easier for everyone. If you find any mistakes or have ideas to improve it, I would appreciate your kind and constructive feedback.

If you enjoyed the book, please share it with those who love to learn!

Please remember that this book is free to share, but its content is copyrighted and cannot be used for commercial purposes without permission.

Follow me to get notified when I publish new content:

- **Website**: marcosborges.phd
- **LinkedIn**: LinkedIn.com/in/MarcosBorgesPhD
- **YouTube**: YouTube.com/@MarcosBorgesPhD

Updated on 8th October 2025.

About Marcos Borges, PhD

Hi, I'm Marcos Borges, a Senior Computer Vision Engineer decoding AI, Machine Learning, Target Tracking, and Sensor Fusion for Autonomous Navigation.

For over two decades, I've navigated the same overwhelming tech landscape you're now, going from selling ice cream to earning a PhD in Applied Mathematics.

I learned what it takes to cut through the noise, build a standout portfolio, and become a highly in-demand specialist.

Now, I'm dedicated to helping engineers move beyond surface-level knowledge so they can build a legacy of impact, not just a career.

Explore my work and projects at marcosborges.phd.

Contents

1 Welcome

Welcome to the Linux Command Line course! Whether you're just starting out or looking to sharpen your skills, this one-hour course covers the core commands and tools you need to use the terminal with confidence.

At some point, every engineer or researcher needs to work in the command line. This course gives you a solid foundation to do just that.

For advanced topics in engineering, AI, or scientific simulations, explore our full range of courses and books at marcosborges.phd.

1.1 Who is this course for?

This course is designed for engineers, researchers, and developers who have a basic familiarity with Linux and core computing concepts. No prior experience with Bash scripting is required.

1.2 How to take this course?

We recommend following along in your own terminal as you progress. The best way to learn the command line is by using it, experiment with each command, try variations, and make it part of your workflow. You can follow the course through the e-book or videos, but consistent, hands-on practice is what builds confidence and skill.

1.3 The Terminal, Command Line, and Shell

The Linux command line is often called the terminal, shell, console, or prompt. While these terms are sometimes used interchangeably, they refer to different parts of the system.

The terminal or console is essentially a text-based interface used to interact with the computer. Its origins go way back to the early 1950s with MIT's Whirlwind I computer, the first to use a typewriter for input and a printer for output.

By the mid-1960s, more advanced display-based terminals like the IBM 2260 began to emerge. During this era, computers were massive machines known as mainframes, and users could connect to it remotely via individual terminals.

Figure 1.1: The IBM 2260 display-based terminal.
Credit: Norsk Teknisk Museum.

These early terminals were quite simple: just a keyboard and a screen. They didn't have the processing power to run programs on their own. Their sole purpose was to send whatever you typed to the central mainframe and then display the data they received back on the screen.

Today, the Linux command line provides a powerful interface for interacting with the computer. Instead of clicking on icons, you type commands into an application called the terminal. Working behind the scenes, a program known as the shell interprets your commands, understands your intent, and instructs the computer to perform the desired actions.

If you're using Linux or macOS, the terminal is already available with common Linux commands. On Windows, install the Windows Subsystem for Linux (WSL). Setup instructions are available at learn.microsoft.com/windows/wsl.

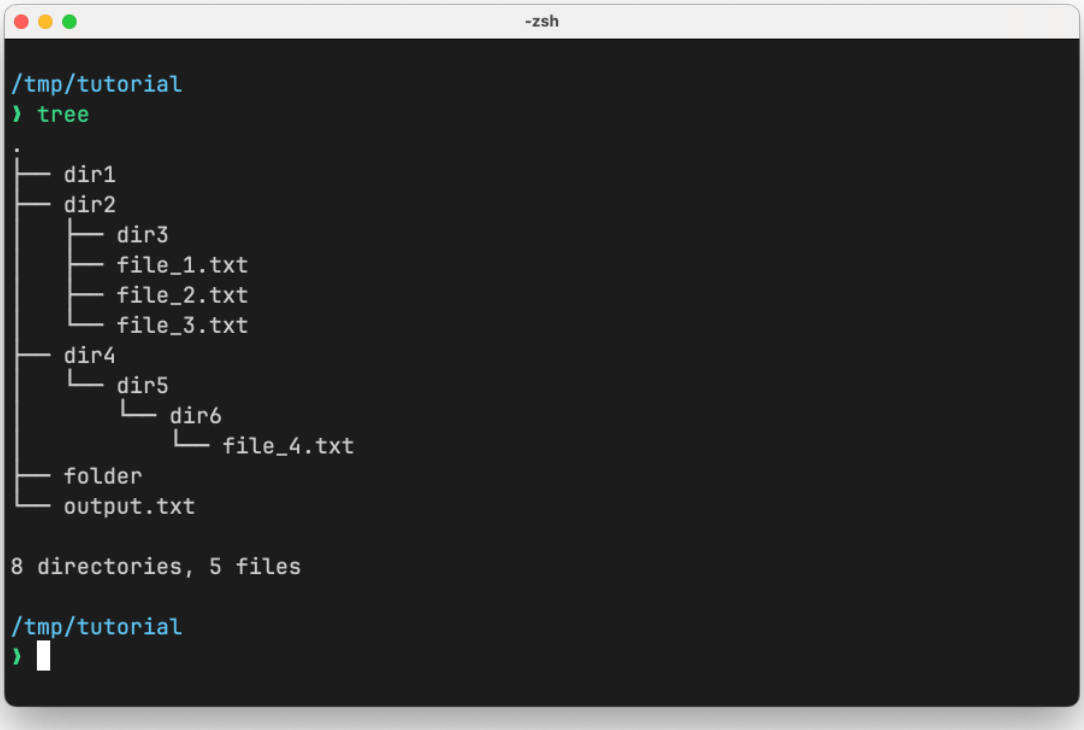


Figure 1.2: macOS Terminal.

2 Getting Started with the Terminal

When you launch the Terminal, you should see a shell prompt similar to this:

```
borges@linux: ~ $
```

Let's begin with the `date` command, which displays the current date and time:

```
borges@linux: ~ $ date
Tue Jul 28 08:25:32 CEST 2025
```

A related command is `cal`, which displays a calendar for the current month:

```
borges@linux: ~ $ cal
     July 2025
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

> **Note**
>
> In some cases, the shell prompt will be omitted to simplify the presentation of commands.

2.1 Navigation

These are the most important commands to help you navigate the file system.

Print the current working directory:

```
pwd
/home/borges
```

> **Note**
>
> If you're using macOS or Windows, the output may differ, so adjust the paths in this course to match the ones shown on your system. Example on macOS:
>
> ```
> pwd
> /Users/borges
> ```

List the contents of the current directory:

```
ls
Desktop  Documents  Downloads  Library  Movies  Music  Pictures
```

List the contents of a specific directory:

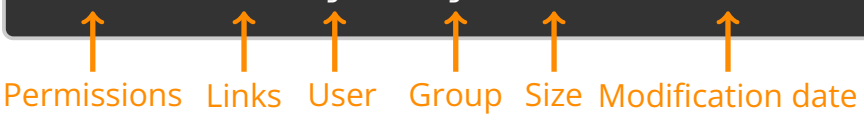
```
ls /home
borges
```

The `ls` command also supports options that modify its output format:

`-a`	Show all content, including hidden ones
`-l`	Show the contents in a long listing format
`-h`	With `-l`, show file size in KB, MB, GB, or TB
`-R`	List subdirectories recursively

The `-l` option displays the contents in a long listing format, showing detailed information such as file permissions, the number of links, the file owner and group, file size in bytes, and the last modification date. Additionally, if the permissions string starts with a `d`, it indicates that the entry is a directory.

```
ls -l /home
total 4
drwxr-x--- 6 borges borges 4096 Jul 28 10:25 borges
```



To make the file size easier to read, add the `-h` option:

```
ls -lh /home
total 4.0K
drwxr-x--- 6 borges borges 4.0K Jul 28 10:25 borges
```

This next command uses the `-a` option to show all files, including hidden ones:

```
ls -alh /home/borges
total 40K
drwxr-x--- 6 borges borges 4.0K Jul 28 08:30 .
drwxr-xr-x 3 root   root   4.0K Jul 28 08:30 ..
-rw-r--r-- 1 borges borges 3.7K Mar 31  2024 .bashrc
-rw-r--r-- 1 borges borges  807 Mar 31  2024 .profile
drwx------ 2 borges borges 4.0K Jul 28 10:25 .ssh
```

The `.` refers to the current directory, `/home/borges` in this example, while `..` refers to its parent directory, that is `/home`. They are not actual files or folders you create, but are built-in directory references used by the filesystem to navigate.

In Linux, hidden files and directories are represented by a dot at the beginning of their names, like the file `.bashrc` and the directory `.ssh`.

Change directory:

```
borges@linux: ~ $ cd /usr/bin
borges@linux: /usr/bin $
```

In this example we start in `~`, which refers to the home directory, and move to `/usr/bin`, as reflected in the shell prompt.

Here are some useful symbols you can use with the `cd` command:

`~`	Home directory
`-`	Previous directory
`.`	Current directory
`..`	Parent directory

From `/usr/bin`, let's navigate to its parent directory:

```
borges@linux: /usr/bin $ cd ..
borges@linux: /usr $
```

Now, our current working directory is `/usr`. Let's return to the home directory:

```
borges@linux: /usr $ cd ~
borges@linux: ~ $
```

From here, we can go back to the previous working directory:

```
borges@linux: ~ $ cd -
borges@linux: /usr $
```

> Mastering these navigation commands is essential for building confidence and efficiency when working with the command line. The more you practice using them, the more natural and intuitive your workflow will become.

2.2 Inspecting Files

This section introduces commands for inspecting files in the file system.

Display file content:

```
cat filename
```

Display the first 10 lines of a file:

```
head filename
```

`-n number`	Show the specified number of lines

Display the last 10 lines of a file:

```
tail filename
```

`-n number`	Show the specified number of lines
`-f`	Follow the file and display new lines as they are added

Count lines, words, characters, and bytes in a file:

```
wc filename
```

`-l, --lines`	Show the number of lines
`-w, --words`	Show the number of words
`-m, --chars`	Show the number of characters
`-c, --bytes`	Show the number of bytes

Let's Practice:

1 Display the contents of a text file:

```
cat /etc/passwd
```

2 Display the first 5 lines of a file:

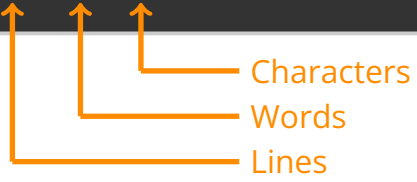
```
head -n 5 /etc/passwd
```

3 Display the last 2 lines of a file:

```
tail -n 2 /etc/passwd
```

4 Count the number of lines, words, and characters in a file:

```
wc /etc/passwd
 32  45 1656 /etc/passwd
```



5 Count only the characters in a file:

```
wc --chars /etc/passwd
```

View file content:

```
less filename
```

`less` is a program for viewing the contents of text files. When a file spans more than one screen, it allows scrolling both forward and backward. Commonly used commands include:

Essential Commands

`h`	Display help menu
`q`	Quit

Movement and Navigation

`j`	Forward one line
`k`	Backward one line
`f`	Forward one window
`b`	Backward one window
`d`	Forward one half-window
`u`	Backward one half-window
`g`	Go to the first window
`G`	Go to the last window

Searching

`/term`	Search forward for `term`
`?term`	Search backward for `term`
`n`	Repeat search, same direction
`N`	Repeat search, opposite direction

Let's Practice:

1 Open a sample text file:

```
less /etc/services
```

2 Practice moving, searching, and quitting:

Scrolling: `j`, `k`, `f`, `b`, `d`, `u`

Searching: `/term`, `n`, `N`

Jumping: `g`, `G`

Quitting: `q`

2.3 Inspecting Commands

A command can be an executable program, an internal shell command, a shell function or script, or an alias that can be defined by several commands.

Display command details:

```
type date

date is /bin/date
```

```
type type

type is a shell builtin
```

Display command location:

```
which type

type: shell built-in command
```

```
which bash

/bin/bash
```

3 Managing Files and Directories

3.1 Copy a file or directory

You can copy a specific **file** or **directory** using the `cp` command:

```
cp source destination
```

> `-r` Copy directories recursively
> `-i` Prompt before overwrite
> `-v` Verbose output

Copy a file to another directory:

```
cp /source_dir/myfile /destination_dir/
```

> The file will retain its original name.

Copy a file to another directory and rename it:

```
cp /source_dir/myfile /destination_dir/new_filename
```

Copy a directory and its content into another directory:

You must use the `-r` flag to copy directories.

```
cp -r /source_dir/mydir /destination_dir/
```

> This copies the entire **mydir** directory and everything within it into **destination_dir**.

3.2 Move or rename a file or directory

The command `mv` is used to move a file or directory from one location to another, or to rename a file or directory within the same location. Unlike `cp`, moving a file or directory **removes** it from the source location.

```
mv source destination
```

> `-i` Prompt before overwrite
> `-v` Verbose output

Rename a file in the current directory:

```
mv old_filename.txt new_filename.txt
```

Move a file to a different directory:

```
mv myfile.txt /destination_dir/
```

> The file retains its original name.

Move a directory to a new location:

```
mv /source_dir/mydir /destination_dir/
```

> This moves the entire **mydir** directory and all its contents to **destination_dir**

3.3 Create a directory

The command `mkdir` (make directory) is used to create one or more new directories in the specified location.

```
mkdir directory_name
```

> `-p` Create parent directories as needed
> `-v` Verbose output

Create a single directory in the current location:

```
mkdir projects
```

Create multiple directories at once:

```
mkdir scripts docs logs
```

Create a nested directory structure:

Use the `-p` option to create multiple directories in a path.

```
mkdir -p project_2025/src/main
```

> This command will successfully create **project_2025**, then **src** inside it, and finally **main** inside **src**.

3.4 Create an empty file

The command `touch` is primarily used to create a new, empty file.

```
touch filename
```

3.5 Remove a file or directory

The command `rm` (remove) is used to permanently delete files or directories. Use this command with caution, as deleted files are typically unrecoverable.

```
rm filename
```

> `-r` Remove directories recursively
> `-f` Force removal without prompt
> `-i` Prompt before each removal

Remove a single file:

```
rm unnecessary_file.txt
```

Remove multiple files:

```
rm log_*.txt temp_file.bak
```

Remove a directory and its contents:

To remove a directory, the `-r` (recursive) flag is required.

```
rm -r old_project_folder/
```

Forcefully remove a directory without being prompted:

```
rm -rf very_old_backup/
```

> **Caution**
>
> The combination `rm -rf` is extremely dangerous.
> Use with extreme caution!

3.6 Working with hard and soft links

In Linux-based systems, links are used to create references or pointers to files or directories, allowing a single item to be accessed via multiple names or locations. We have two distinct kinds of links:

Hard link is a second name for the same file data on disk. If one name is deleted, the other still works.

Soft link (symbolic link) is a shortcut that points to another file or directory. If the target is deleted, the link breaks.

Create a hard link:

```
ln target link_name
```

> Hard links can only be created for files, not directories.

Create a symbolic (soft) link:

```
ln -s target link_name
```

> Soft links can be created for files and directories.

Let's Practice:

1 Create a playground directory:

```
mkdir /tmp/playground
```

2 Change to the playground directory:

```
cd /tmp/playground
```

3 Create multiple directories following the example:

```
dir1
dir2
|---- dir3
dir4
|---- dir5
      |---- dir6
folder
```

```
mkdir dir1 dir2 dir4 folder
```

```
mkdir dir2/dir3 dir4/dir5
```

```
mkdir dir4/dir5/dir6
```

You can create all directories at once using a single command:

```
mkdir -p dir1 dir2 dir2/dir3 dir4 dir4/dir5/dir6 folder
```

4 Create files inside `dir3`:

```
dir2
|---- dir3
      |---- file_1.txt
      |---- file_2.txt
      |---- file_3.txt
```

```
touch dir2/dir3/file_1.txt
```

```
touch dir2/dir3/file_2.txt
```

```
touch dir2/dir3/file_3.txt
```

5 Recursively list all directories and files:

```
ls -R
```

6 Copy `/etc/passwd` into `dir6/file_4.txt`:

```
dir4
|---- dir5
      |---- dir6
            |---- file_4.txt
```

```
cp /etc/passwd dir4/dir5/dir6/file_4.txt
```

7 Create a symbolic link to `file_4.txt`:

```
dir4
|---- dir5
      |---- dir6
            |---- file_4.txt

soft_link.txt -> dir4/dir5/dir6/file_4.txt
```

```
ln -s dir4/dir5/dir6/file_4.txt soft_link.txt
```

8 List files in the current directory with detailed info:

```
ls -l
```

9 Create a hard link to `file_4.txt`:

```
dir4
|---- dir5
      |---- dir6
            |---- file_4.txt

hard_link.txt
```

```
ln dir4/dir5/dir6/file_4.txt hard_link.txt
```

10 Remove the `file_4.txt` file:

```
dir4
|---- dir5
      |---- dir6
            |---- file_4.txt
```

```
rm dir4/dir5/dir6/file_4.txt
```

11 List files in the current directory with detailed info:

```
ls -l
```

> Notice that the symbolic link `soft_link.txt` is now broken, while the hard link `hard_link.txt` remains fully accessible.

12 Attempt to display the content of the `soft_link.txt` file:

```
cat soft_link.txt
```

> The file no longer exists because the link is broken.

13 Display the first 10 lines of `hard_link.txt` file:

```
head hard_link.txt
```

14 Rename the hard link `hard_link.txt` to `output.txt`:

```
mv hard_link.txt output.txt
```

15 Remove the `soft_link.txt` file:

```
rm soft_link.txt
```

16 Recursively list all directories and files:

```
ls -R
```

```
dir1
dir2
|---- dir3
      |---- file_1.txt
      |---- file_2.txt
      |---- file_3.txt
dir4
|---- dir5
      |---- dir6
folder
output.txt
```

4 Managing IO and Errors

This section covers commands for controlling the standard input, output, and error of processes, making it easier to manage and manipulate data.

In Linux, every process automatically starts with three data streams:

1. **Standard input (stdin)**: the channel through which a program receives data, usually from the keyboard or a file.
2. **Standard output (stdout)**: where a program sends its results, usually shown on the terminal.
3. **Standard error (stderr)**: used to display error messages.

A process is simply a program that is running on your computer. When you start an application or run a command, the system creates a process for it, which includes the program's code, its current activity, and the resources it needs, like memory or CPU time. In short, a process is the active, running instance of a program.

4.1 Redirecting the standard output

To redirect the standard output of a process to a file instead of displaying it on the screen, use the redirection operator `>` followed by the file name.

For example, we can redirect the output of the `cal -m July` command to a file instead of displaying it on the screen:

```
cal -m July > /tmp/calendar.txt
```

Then, we can display the file content:

```
cat /tmp/calendar.txt
     July 2025
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Now, let's see what happens if we type the command incorrectly:

```
cal -m month > /tmp/calendar.txt
cal: month is neither a month number (1..12) nor a name
```

> We get an error message because the command expects either a valid month name or a number between 1 and 12, not the term month.

Now, let's display the file content:

```
cat /tmp/calendar.txt
```

> The file is empty because the command `cal -m month` produced an error and didn't generate the calendar as expected.

As you may notice, the redirection operator `>` overwrites the file, discarding its previous data. If you want to preserve the existing data, you can use the append operator `>>`, which adds new data to the end of the file.

Overwriting the file with the July calendar:

```
cal -m July > /tmp/calendar.txt
```

Appending the file with the August calendar:

```
cal -m August >> /tmp/calendar.txt
```

Displaying the contents of the file:

```
cat /tmp/calendar.txt
     July 2025
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31

    August 2025
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

4.2 Redirecting the standard error

Redirecting the standard error is less intuitive than redirecting the standard output. Internally, the shell refers to the standard input, output, and error using file descriptors 0, 1, and 2 respectively:

- **File descriptor 0**: Standard input (stdin)
- **File descriptor 1**: Standard output (stdout)
- **File descriptor 2**: Standard error (stderr)

Let's try a command that generates a normal output and an error at the same time:

```
ls myfile /usr
ls: cannot access 'myfile': No such file or directory
/usr:
bin games include lib libexec local sbin share src
```

Redirect the output to a file and the error to the terminal:

```
ls myfile /usr > /tmp/output.txt
ls: cannot access 'myfile': No such file or directory
```

> Try viewing the contents of the output file to see the results.

Redirect the output to the terminal and the error to a file:

```
ls myfile /usr 2> /tmp/output.txt
/usr:
bin games include lib libexec local sbin share src
```

Redirect both the output and the error to a file:

```
ls myfile /usr &> /tmp/output.txt
```

There is also a traditional way to do it:

```
ls myfile /usr > /tmp/output.txt 2>&1
```

> Here we perform two redirections: First we redirect the standard output to a file, then we redirect the standard error (file descriptor 2) to the standard output (file descriptor 1).

Discard the standard error:

```
ls myfile /usr 2> /dev/null
/usr:
bin games include lib libexec local sbin share src
```

> The `/dev/null` file is a special device that acts like a bit bucket or black hole. Any data written to it is immediately discarded. It is a virtual device with no physical storage, commonly used to suppress command output and errors, create empty files, or provide an empty input stream to programs.

4.3 Redirecting the standard input

Standard input redirection allows a command to receive data from a file or variable instead of the keyboard, using the `<` operator.

Let's create a simple text file:

```
printf "Zucchini\nApple\nCherry\n" > /tmp/items.txt
```

> The `printf` command is used to format and print data. In the example above, we redirect the standard output to a file.

Redirect the standard input from a file:

```
sort < /tmp/items.txt
Apple
Cherry
Zucchini
```

4.4 Pipeline

A pipeline allows you to use the pipe operator `|` to send the standard output of one command as the standard input to another command.

Let's display some data and pass it through a pipeline to sort the output:

```
printf "Zucchini\nApple\nCherry\n" | sort
Apple
Cherry
Zucchini
```

Now, let's sort some data and use a pipeline to remove duplicates:

```
printf "Red\nBlue\nYellow\nBlue\nYellow\nRed" | sort | uniq
Blue
Red
Yellow
```

> The `uniq` command reports or filters out repeated lines in a file. It only works on sorted input.

Here's an example of filtering by the specific term **python**:

```
ls /usr/bin | grep python
pybabel-python3
python3
python3.14
```

> The `grep` command is used for searching text patterns in files. Don't worry about understanding everything for now, as we will explore it in more detail in later chapters.

Explore the docs for each command. We have only scratched the surface!

With more practice you will see how redirection is used for solving real problems, and how standard input, output, and error are at the core of nearly every command-line tool.

5 Searching files

The `find` command is a powerful tool for searching files and directories based on their name, type, permissions, date, ownership, size, and more. It can also execute commands on the results, making it very flexible.

```
find [path] [expression]
```

Common paths

`/`	Root directory (entire system)
`.`	Current directory
`~`	Home directory

Common expressions

`-name`	Filter by file name
`-type d`	Filter by directories
`-type f`	Filter by regular files
`-type l`	Filter by symbolic links
`-user`	Filter by owner
`-group`	Filter by group
`-size`	Filter by file size

Sizes

`c`	Bytes
`k`	Kilobytes
`M`	Megabytes
`G`	Gigabytes

5.1 Search for files by name

Searching for any regular file named **bash**:

```
find /usr/ -name "bash"
```

Searching for files ending with the term **bash**:

```
find /usr/ -name "*bash"
```

Searching for files starting with the term **bash**:

```
find /usr/ -name "bash*"
```

Searching for files containing the term **bash**:

```
find /usr/ -name "*bash*"
```

Searching for files with a **.txt** extension:

```
find /usr/ -name "*.txt"
```

5.2 Search for files by type

Searching for directories:

```
find /usr/ -type d
```

Searching for directories named **linux**:

```
find /usr/ -type d -name "linux"
```

Searching for regular files:

```
find /usr/ -type f
```

Searching for `.txt` files with names starting with **bash**:

```
find /usr/ -type f -name "bash*.txt"
```

Searching for symbolic links:

```
find /usr/ -type l
```

Searching for symbolic links named **python3**:

```
find /usr/ -type l -name "python3"
```

5.3 Search for files by size

Searching for files larger than 10 Megabytes:

```
find /usr/ -size +10M
```

Searching for files smaller than 1 Kilobyte:

```
find /usr/ -size -1k
```

5.4 Search for files by owner

Searching for files owned by your user:

```
find ~ -user "$(whoami)"
```

> The `whoami` command returns your username. To run a command inside a string we use `$( command )`.

5.5 Executing commands with -exec

The `-exec` option allows you to run a command on each file found. This makes `find` much more powerful. The syntax is:

```
find [path] [expression] -exec command {} \;
```

The `{}` is replaced by the current file, and the sequence must end with `\;`.

Examples:

Listing details of all `.txt` files:

```
find ~ -name "*.txt" -exec ls -l {} \;
```

Printing the first line of each `.txt` file:

```
find ~ -name "*.txt" -exec head -n 1 {} \;
```

Removing empty files (safe to try in a test directory):

```
find ./testdir -type f -empty -exec rm {} \;
```

> **Warning**
>
> This command is safe because of the `-empty` parameter.
> When using `-exec rm`, be extra careful not to delete important files.

6 Searching text

The `grep` command is used to search text inside files using patterns and regular expressions. It is one of the most common tools for quickly finding matching lines in logs, source code, or any other text file.

```
grep [options] pattern [file...]
```

Common options

`-i`	Ignore case (case-insensitive search)
`-r`	Search recursively in directories
`-n`	Show line numbers of matches
`-v`	Invert match (show non-matching lines)
`-E`	Use extended regular expressions
`-l`	Show only names of files with matches
`-c`	Count the number of matches

Basic regular expressions

`^word`	Match lines starting with `word`
`word$`	Match lines ending with `word`
`.`	Match any single character
`.*`	Match zero or more of any character
`[abc]`	Match one character: `a`, `b` or `c`
`[0-9]`	Match any digit
`word1|word2`	Match either `word1` or `word2`

6.1 Search for a simple word

Searching for the word **error** in a file:

```
grep "error" logfile.txt
```

Ignoring case (matches error, Error, ERROR, etc.):

```
grep -i "error" logfile.txt
```

6.2 Search using anchors

Searching for lines starting with **root**:

```
grep "^root" /etc/passwd
```

Searching for lines ending with **bash**:

```
grep "bash$" /etc/passwd
```

6.3 Search with character classes

Searching for lines that contain a number:

```
grep "[0-9]" data.txt
```

Searching for lines containing **cat**, **bat**, or **hat**:

```
grep "[cbh]at" words.txt
```

6.4 Extended regular expressions

With `-E` we can use more advanced patterns.

Searching for either **dog** or **cat**:

```
grep -E "dog|cat" animals.txt
```

Searching for words ending with **ing**:

```
grep -E "[a-zA-Z]+ing" notes.txt
```

6.5 Counting and file matches

Counting the number of matches for the word **error**:

```
grep -c "error" logfile.txt
```

Showing only the names of files containing the word **main**:

```
grep -rl "main" ~/projects/
```

7 Introduction to Bash scripting

A Bash script is a text file containing commands that are executed in sequence. Scripts are essential for system administration, allowing you to automate tasks, create simple programs, and combine existing `commands` into efficient, reusable workflows.

7.1 Your first Bash script

1 Create a script file using nano or your favorite code editor:

```
nano /tmp/script.sh
```

2 Copy and paste the content below, and save:

```
#!/usr/bin/env bash
# A simple script
echo "Hello, world!"
```

The first line, known as the **shebang**, specifies Bash as the **interpreter** for the script. The second line is a comment. The third line uses the `echo` command to print the specified string to the standard output.

3 Make the file executable:

To run a bash script correctly, you must ensure it has execution permission. For that, we use the `chmod` command (short for **ch**ange **mod**e), which is a fundamental Linux command used to control access to files and directories.

Close the nano editor and set the execution permission to the script:

```
chmod +x /tmp/script.sh
```

4 Run the script:

```
bash /tmp/script.sh
```

You've done great work! With a little more creativity, you'll be able to create incredible things using Bash scripting.

7.2 Comments

Comments help explain the purpose of certain lines. They should only be used to explain less obvious parts of the code.

```
# This is a comment!
# Comments start with a hash (#)
#
# Something important here! Albert Einstein
```

7.3 Variables

Variables are used to store values. No spaces are allowed around the equal sign.

```
NAME="Alice"
CITY="Paris"

echo "${NAME} lives in ${CITY}"
```

> Use `${VAR_NAME}` to access the value of a variable. Always quote variables like `"${VAR_NAME}"` to avoid unexpected word splitting.

7.4 Multi-line variables

You can assign multi-line text to a variable using the **here-document** syntax, which allows you to embed a block of text or commands directly within a script.

Below is an example of how you can define a multi-line variable.

Copy and paste this into the terminal:

```
TEXT="$(cat << EOF
The current directory is: ${PWD}
You are logged in as: $(whoami)
EOF
)"

echo "${TEXT}"
```

By default the shell performs parameter expansion, command substitution inside the here-document. For example, `${PWD}` expands to the current directory and `$(whoami)` is replaced by the current user.

If you need to preserve the literal contents disabling expansions and substitutions, quote the opening delimiter `<< 'EOF'`, so the text is stored or printed exactly as written.

Below is an example of how to define a multi-line variable without expansion. Pay attention to the details and compare it with the previous (unquoted) example to observe the difference.

Copy and paste the following into the terminal:

```
TEXT="$(cat << 'EOF'
Example without expansion using quoted 'EOF'
The current directory is: ${PWD}
You are logged in as: $(whoami)
EOF
)"

echo "${TEXT}"
```

> Using `<< 'EOF'` keeps the text literal. Without the quotes, variables inside will be expanded.

7.5 Arrays

Bash supports indexed arrays, which provide a way to store multiple data elements that can be accessed by their position (index), starting from 0. Check the example below, and you'll understand it easily.

Copy and paste this into the terminal:

```
# Enable KSH_ARRAYS in Zsh to make array indexing start at 0
setopt KSH_ARRAYS

COLORS=("red" "green" "blue")

echo "Color index 0: ${COLORS[0]}"
echo "Color index 1: ${COLORS[1]}"
echo "Color index 2: ${COLORS[2]}"
```

7.6 Reading input from the user

The command `read` is used to capture input from the user and store it in a variable, in this example `USERNAME`. When this command runs, the script pauses and waits for the user to type something and press Enter. Whatever the user types is then assigned as the value of the variable, allowing the script to use that input later in the program.

```
echo "Enter your name:"
read USERNAME

echo "Hi ${USERNAME}!"
```

Let's Practice:

1 Create a new script:

```
nano /tmp/calendar.sh
```

2 Copy and paste the content below, and save:

```
#!/usr/bin/env bash
echo "Calendar Display Utility"
echo
echo "Please enter the month number (1-12):"

read MONTH

# Print the calendar for the month
cal -m "${MONTH}"
```

3 Make the file executable:

```
chmod +x /tmp/calendar.sh
```

4 Run the script:

```
bash /tmp/calendar.sh
```

7.7 Control flow

Control flow in Bash scripting is essential for making decisions and performing repetitive tasks. It determines the order in which commands are executed based on conditions.

Mastering control flow allows a script to move beyond simple sequential commands to dynamic and powerful automation.

7.8 If statements

The if statement is a control structure that allows to execute one or more instructions based on the evaluation of a command or an expression called a condition.

Here is an example of if statement:

```
if [[ "CONDITION" ]]; then
    "INSTRUCTIONS"
fi
```

Now, a more complete example:

```
NAME="Alice"

if [[ "${NAME}" = "Alice" ]]; then
  echo "Welcome, Alice!"
else
  echo "You are not Alice."
fi
```

7.9 Tests and conditions

We have several ways to perform tests and conditional checks in Bash. Below are the most commonly used tests:

`[[ -f FILE ]]`	True if file exists and is a regular file
`[[ -d DIR ]]`	True if directory exists
`[[ -z STR ]]`	True if string is empty
`[[ -n STR ]]`	True if string is not empty
`[[ NUM1 -eq NUM2 ]]`	True if numbers are equal
`[[ NUM1 -lt NUM2 ]]`	True if NUM1 is less than NUM2
`[[ NUM1 -gt NUM2 ]]`	True if NUM1 is greater than NUM2

Check if a file exists:

```
if [[ -f "myfile.txt" ]]; then
  echo "File exists"
else
  echo "File not found"
fi
```

Compare two numbers:

```
if [[ 1 -lt 2 ]]; then
  echo "Are you sure?"
fi
```

7.10 For loops

A for loop allows a block of code to be executed repeatedly. For example, we can use it to print all the elements of an array like this:

```
COLORS=("red" "green" "blue")

for COLOR in "${COLORS[@]}"
do
  echo "Color: ${COLOR}"
done
```

Here is another way to do it using loop over strings:

```
for COLOR in "red" "green" "blue"
do
  echo "Color: ${COLOR}"
done
```

You can also loop over a range:

```
for i in {0..3}
do
  echo "Number: ${i}"
done
```

8 This is just the beginning

In the preceding chapters, we demystified the Linux Command Line, giving you the power to manipulate files and automate tasks.

The only thing left to do now is practice, practice, and practice.

Even though we covered a lot of ground in our adventure, we barely scratched the surface as far as the command line goes. There are still thousands of command line programs left to be discovered and enjoyed.

This is just the beginning of your journey. May the Force be with you!

Level up your career in Tech

Get notified when I publish new content.

Join my free newsletter packed with actionable insights on AI, Machine Learning, Target Tracking, and Sensor Fusion for Autonomous Navigation.

Gain access to exclusive tips and strategies I don't share anywhere else, and learn the skills top employers are hiring for.

Subscribe for free at marcosborges.phd.