

# BASH

# CHEAT SHEET

created by  
Tomi Mester



I originally created this cheat sheet for my 6-week data science online course participants.\* But I have decided to open-source it and **make it available to everyone who wants to learn bash and the command line.**

It's designed to give you a meaningful structure but also to let you add your own notes (that's why the empty boxes are there). **It starts from the absolute basics (cd ..) and includes everything that you will need as a junior data analyst/scientist (commands, scripting, automations, etc.).**

The ideal use case of this cheat sheet is that you print it in color and keep it next to you while you are learning and practicing Bash on your computer.

Enjoy!

Cheers,  
Tomi Mester



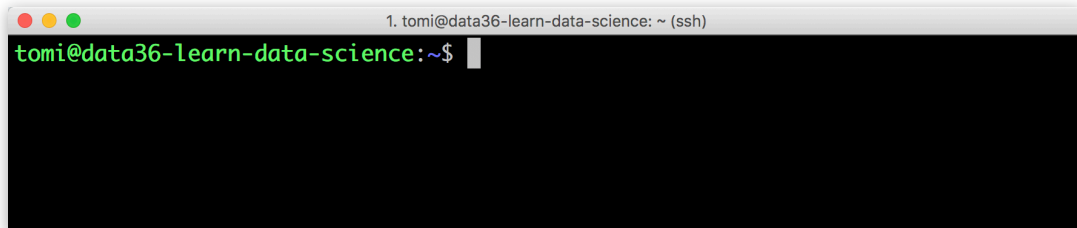
\*The online courses I mentioned:

**Online command line and bash tutorial (free):** [data36.com/bash-tutorial](https://data36.com/bash-tutorial)  
**6-week Data Science course:** [data36.com/jds](https://data36.com/jds)

## THE PROMPT

When working in the command line, at the beginning of every line you will see the prompt.

Example:

A terminal window with a title bar that reads "1. tomi@data36-learn-data-science: ~ (ssh)". The terminal content shows a green prompt "tomi@data36-learn-data-science:~\$" followed by a white cursor bar.

tomi@data36-learn-server:~\$

It's built from six elements:

- your user name (e.g. tomi)
- @
- your server name (e.g. data36-learn-server)
- :
- folder name (e.g. ~ which means your user's home directory)
- \$

Your commands go after the \$ sign.

## ABOUT THE COMMAND LINE COMMANDS

In the command line, you interact with your server exclusively by typing commands. You don't have a cursor or a graphical user interface.

A few commonalities of bash commands:

- They are usually created to do one simple thing and do that one thing very well.
- The name of the commands are almost always the short form of the things they do. (e.g. **pwd** stands for "**p**rint **w**orking **d**irectory".)
- Their original function can be modified with options. (More details later.)

[your notes]

## BASIC DIRECTORY COMMANDS

### `pwd`

Prints the name of your current working directory.

### `ls`

Lists all the content of your current working directory.

### `cd`

You move from your current working directory to your user's home directory.

Note: **cd** stands for **c**hange **d**irectory.

### `cd test_dir`

You move to the `test_dir` directory - if `test_dir` exists in your current working directory.

### `cd ..`

Moves up one folder.

### `cd -`

Moves to the folder where you previously were.

### `mkdir one_more_dir`

Creates a new directory called `one_more_dir`.

## **BASIC FILE COMMANDS**

```
touch test_file.txt
```

Creates an empty file called test\_file.txt.

```
cp test_file.txt test_file_copy.txt
```

Copies your original test\_file.txt and names the new copy to test\_file\_copy.txt.

```
cp test_file.txt one_more_dir/test_file.txt
```

Copies your original test\_file.txt into the one\_more\_dir directory - and keeps its original name. (Only works if one\_more\_dir is in your current working directory.)

```
mv test_file.txt one_more_dir/test_file.txt
```

Moves your original test\_file.txt into the one\_more\_dir directory.  
(Only works if one\_more\_dir is in your current working directory.)

```
mv test_file.txt test_file_some_new_name.txt
```

Renames your original test\_file.txt to test\_file\_some\_new\_name.txt.

```
rm test_file.txt
```

Removes the test\_file.txt file - if it exists.

Note: Be careful! In bash, if you delete a file, it will be deleted for good. There's no "Trash" folder or "Restore" button in case you remove something by accident.

More about the basics: <https://data36.com/bash-basics>

## TIPS AND TRICKS

### **#1**

#### **clear**

Clears your terminal screen.

### **#2**

You can use the up (↑) and down (↓) arrows on your keyboard to bring back your previous commands.

### **#3**

#### **history**

Brings back all your previously typed-in commands and prints them to your screen.

### **#4**

Hit the TAB key (→) on your keyboard to auto-extend your typed-in text (e.g. commands or file names.)

More tips and tricks: <https://data36.com/bash-tricks>

## PRINTING

```
echo "Hello, World!"
```

Prints Hello, World! to your screen.

```
cat some_data.csv
```

Prints the entire contents of the `some_data.csv` file to your screen.

```
head some_data.csv
```

Prints the first ten rows of the `some_data.csv` file to your screen.

```
head -50 some_data.csv
```

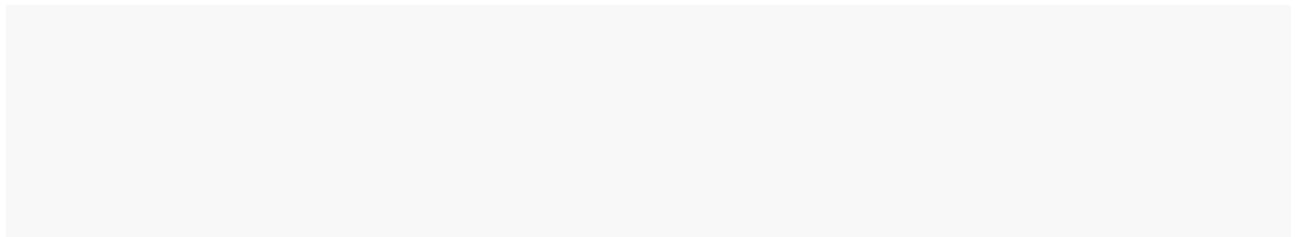
Prints the first fifty rows of the `some_data.csv` file to your screen.

```
tail some_data.csv
```

Prints the last ten rows of the `some_data.csv` file to your screen.

```
tail -50 some_data.csv
```

Prints the last fifty rows of the `some_data.csv` file to your screen.



## COUNTING

```
wc some_data.csv
```

Counts the number of lines, words and characters in `some_data.csv` - and prints this information to the screen.

```
wc -l some_data.csv
```

Counts only the number of lines in `some_data.csv`. (Note: **-l** stands for **l**ine.)

```
wc -w some_data.csv
```

Counts only the number of words in `some_data.csv`. (Note: **-w** stands for **w**ord.)

```
wc -c some_data.csv
```

Counts only the number of characters in `some_data.csv`.

## OPTIONS

By using an option, you can modify what your original command does. Most commands have plenty of options. You can apply them by:

1. typing your original command (e.g. `wc`)
2. then a space and a dash ( `-` )
3. then the option name (e.g. `w`)

General syntax:

**command -option**

A few examples:

**`wc -w some_data.csv`**

On its own, `wc` counts the number of lines, words and characters in `some_data.csv`.  
But with the `-w` option, it only counts words.

**`ls -l`**

`ls` lists all the content of your current working directory.

But with the `-l` option, it also displays additional information for the given files and directories (e.g. size, date of creation, owner, permissions).

**`tail -50 some_data.csv`**

`tail` prints the last 10 rows of the `some_data.csv` file.

But with the `-50` option, it prints the last 50 rows instead.



## MANUALS (LIST OF OPTIONS FOR EVERY COMMAND)

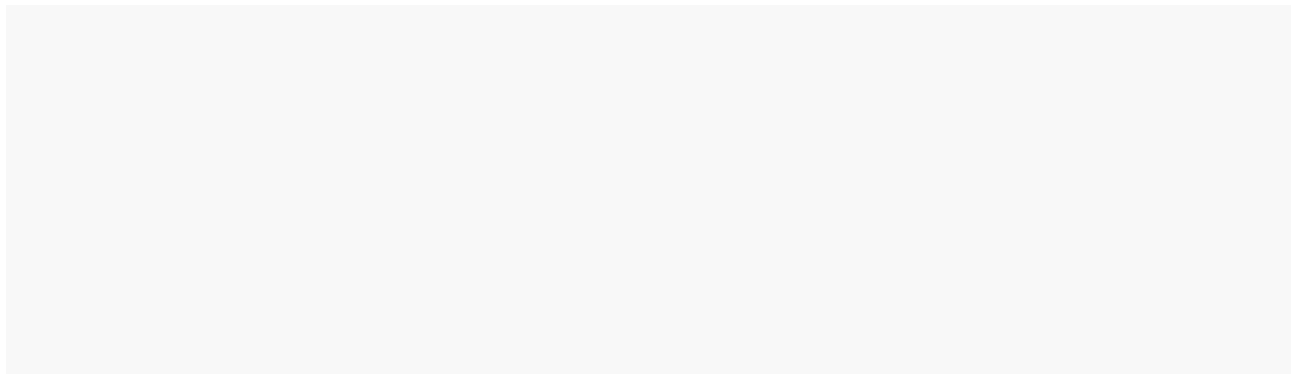
There is a built-in manual for every command, which you can open directly from the command line.

Example:

```
man wc
```

Opens the manual for the `wc` command. There you'll find a general description of the command and the descriptions for all the options it has.

When you are finished, you can exit the manual by hitting the **Q** key on your keyboard.



## PRINTING TO FILE

In complex data projects, you don't want to print results to your terminal screen. You want to save them into files, so you can reuse them later.

Examples:

```
head some_data.csv > first_ten_rows.csv
```

Takes the first ten rows of the `some_data.csv` file and prints them into the file called `first_ten_rows.csv`. (If the `first_ten_rows.csv` file didn't exist before, it will be created. If it did exist, then it will be overwritten.)

```
head some_data.csv >> some_rows.csv
```

Takes the first ten rows of the `some_data.csv` file and adds them to the end of the file called `some_rows.csv`. (If the `some_rows.csv` file didn't exist before, it will be created. If it did exist, then the new lines will be appended to the file after the already-existing lines.)

More about options and printing to file: <https://data36.com/bash-concepts>

## FILTERING

```
cut -d';' -f1 some_data.csv
```

Instead of the whole file, it prints only the first column of the some\_data.csv file. The field separator (aka delimiter) between the columns is a semicolon (;).

```
cut -d'-' -f1,5,12,13 some_data.csv
```

Prints the first, fifth, twelfth and thirteenth columns of the some\_data.csv file. The delimiter is a dash (-).

```
grep "something" some_data.csv
```

Returns all the lines in the some\_data.csv file that contain the "something" string. (It's case sensitive by default.)

```
grep -v "something" some_data.csv
```

Returns all the lines in the some\_data.csv file that do not contain the "something" string. (It's case sensitive by default.)

```
grep -i "Something" some_data.csv
```

Returns all the lines in the some\_data.csv file that contain the "Something" string. It's not case sensitive.

More about grep and cut: <https://data36.com/grep-cut>

## SORTING AND UNIQUE VALUES

```
sort some_data.csv
```

Sorts the some\_data.csv file by alphabetical order (by default) and prints the result to your screen.

```
sort -n some_data.csv
```

Sorts the some\_data.csv file by numerical order.

```
sort -r some_data.csv
```

Sorts the some\_data.csv file by alphabetical order (by default) and in reverse.

```
sort -r -n some_data.csv
```

Sorts the some\_data.csv file by numerical order and in reverse.

```
sort -u some_data.csv
```

Sorts the some\_data.csv file and removes duplicates. (Note: **-u** stands for **u**nique.)

```
sort -n -t',' -k2 some_data.csv
```

Sorts the some\_data.csv file in numerical order -- and the column that it uses for sorting is the second column. The field separator is a comma (,).

```
uniq some_data.csv
```

Unifies repeated lines that follow each other.

Note: `sort -u` removes all duplicated rows. `uniq` removes only those duplicated rows that follow each other.

```
uniq -c some_data.csv
```

Unifies repeated lines that follow each other and counts the number of occurrences.

More about sort and uniq: <https://data36.com/sort-uniq>

## PIPES - USING MULTIPLE COMMANDS

In complex data projects, one bash command is often not enough to do the job. You'll need to apply several commands after each other. In theory, you could do that by creating temporary files as the outputs to reuse them in the next commands as inputs... But that'd be repetitive and way too complicated.

To get things done more simply and cleanly, you can use pipes instead.

A pipe takes the output of your command and redirects it directly into your next command.

Examples:

```
head -50 some_data.csv | tail -10
```

Takes the first 50 rows of the `some_data.csv` file. The pipe redirects this output into `tail`, which prints the last 10 rows of those first 50 rows. As a result, you will have the rows from 41 to 50.

```
sort some_data.csv | uniq -c
```

Sorts the `some_data.csv` file and then it counts all occurrences of every unique row.

```
cut -d';' -f1 some_data.csv | grep "something" | wc -l
```

Takes the first column of the `some_data.csv` file. Then it filters for the rows that contain the string "something". Eventually it counts the lines. As a result, you will have the number of lines that contain the string "something" in the first column of the file.

More about pipes: <https://data36.com/bash-concepts>

## BASH SCRIPTING

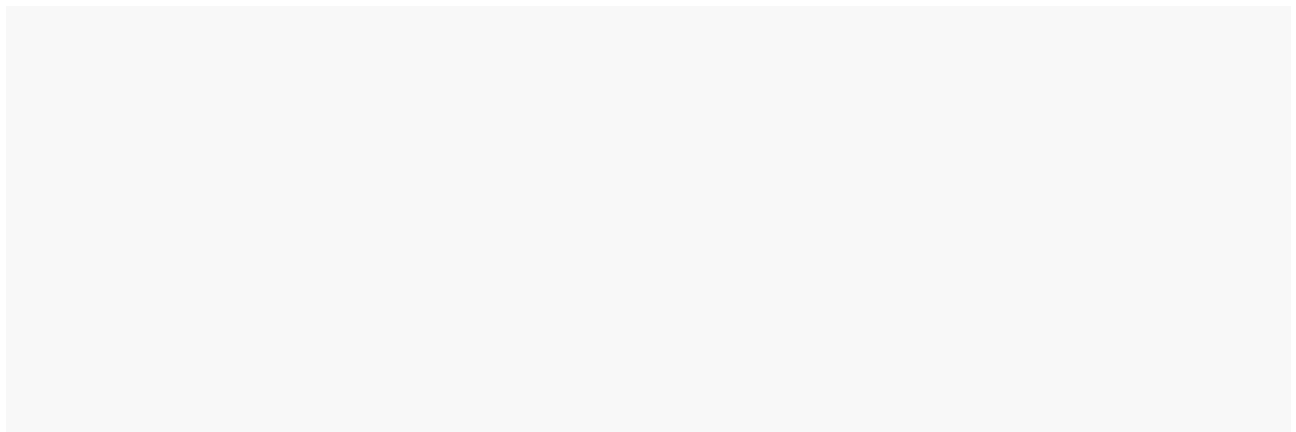
### **mcedit**

Opens a text editor in bash.

(If you haven't set up `mcedit` yet, go here: <https://data36.com/server-setup>)

In the editor, you can create a bash script by simply adding commands listed one after another (the same commands that you would type on the command line).

When you run this script, the commands in it will be executed one by one.



Creating and running a bash script step by step:

1. `mcedit` (it opens your text editor)
2. In the editor, add your commands line by line!  
Example:  

```
echo "number of lines in file:"  
wc -l some_data.csv  
echo "number of characters in file:"  
wc -c some_data.csv
```
3. Still in the editor, add this line to the very beginning of your bash script:  

```
#!/usr/bin/env bash
```

(This line is called the shebang. It tells Ubuntu that your script is in bash.)
4. Click "**10 Quit**" in the bottom right corner of the editor and save your file.
5. Name your file to `demo_script.sh` (or whatever you prefer)!
6. Give permission to yourself to run your script. Type this command:  

```
chmod 700 demo_script.sh
```
7. Run your script by typing this on the command line:  

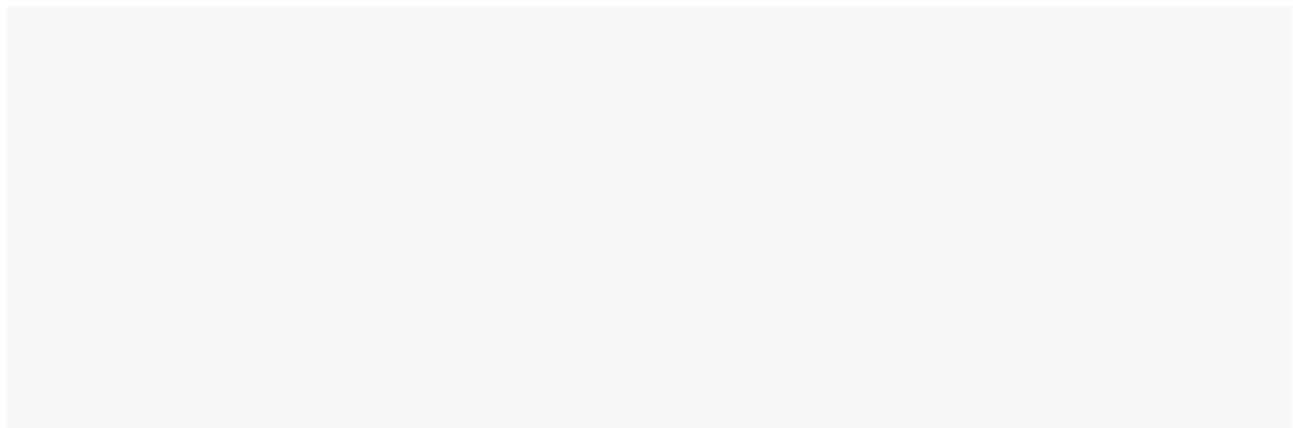
```
./demo_script.sh
```

Done!

After running the script, your output will be something like this:

```
number of lines in file:  
672  
number of characters in file:  
11424
```

You can create various bash scripts (even hundreds of lines long) and even Python or SQL scripts the same way.



## **AUTOMATIONS**

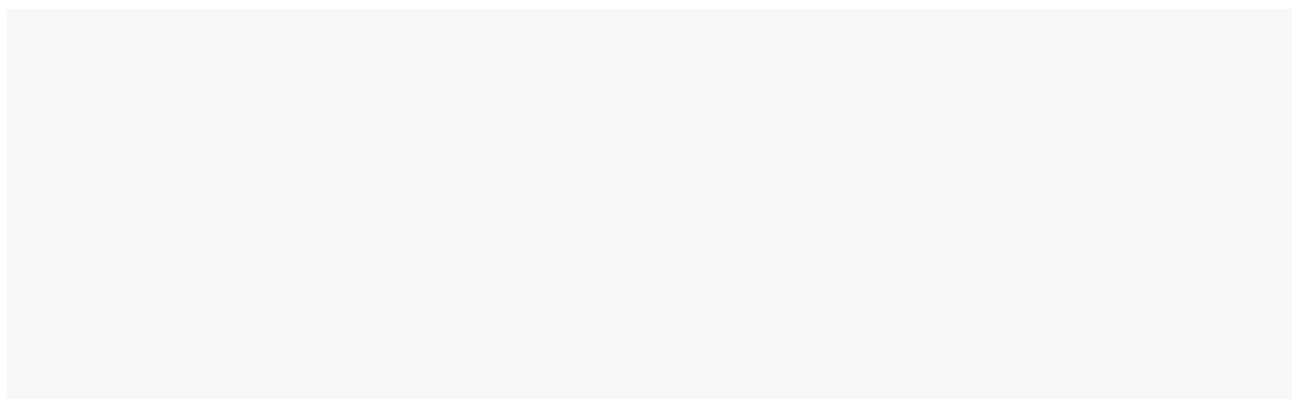
You can schedule your scripts (and your bash commands) to run automatically on your remote server every day, every hour, even every minute.

For that, you'll have to use cron.

You can configure cron by opening crontab. Type this:

```
crontab -e
```

It will open a text file which starts with a detailed manual of how it works. After the manual, you can add your actual tasks to schedule.



One line is one task and you have to define 6 parameters in each line:

- The minute of the hour
- The hour of the day
- The day of the month
- The month
- The day of the week
- The command you want to run.

The first five parameters define the when - the 6th defines the what.

Example:

```
01 00 * * * /home/demo_user/demo_script.sh
```

If you have this line in your crontab, it will run your `demo_script.sh` script (that's located in the `/home/demo_user/` folder) every day, one minute after midnight.

Here's a 9-minute detailed video tutorial about bash automations in crontab:

<https://data36.com/automation-crontab>

## VARIABLES

In bash, you can assign a value to a variable as simply as:

```
variable_name=variable_value
```

Note: If you assign a new value to a variable that you have used before, it will overwrite your previous value.

Examples:

```
a=100  
b='hello, world'  
c=true  
d=0.75
```

You can refer to any of these variables by typing a **\$** sign and the variable name itself.

Examples:

```
echo $a  
Prints 100.
```

```
echo $b  
Prints hello, world.
```

```
echo $c  
Prints true.
```

```
echo $d  
Prints 0.75.
```



## IF STATEMENTS

If statements are great for evaluating a condition and taking certain action(s) based on the result.

Example:

```
a=10
b=20
if [[ $a == $b ]]
then
echo 'yes'
else
echo 'no'
fi
```

## WHILE LOOPS

While loops are great for repeating and executing one (or more) commands until a condition is fulfilled.

Example:

```
i=0
while [ $i -lt 10 ]
do
echo $i
i=$((i + 1))
done
```

More about variables, if statements and while loops in bash:

<https://data36.com/bash-if-while>

## A FEW MORE USEFUL COMMANDS

### **sed**

sed was created to parse and transform text in the command line.

```
sed 's/,/;/g' some_data.csv
```

Replaces all comma (,) characters with semicolons (;) in the some\_data.csv file and it prints the result to the screen.

### **date**

Prints the current time and date.

(Example output: Sat Aug 10 23:21:55 UTC 2019)

```
date +%D
```

Prints the current date in mm/dd/yy format. (Example output: 08/10/19)

```
date +%Y-%m-%d
```

Prints the current date in yyyy-mm-dd format. (Example output: 2019-08-10)

### **awk**

awk is another great command line tool for text processing and data cleaning.

```
awk -F',' '{print $3,$2,$5}' some_data.csv
```

Prints the third, second and fifth columns (in that particular order) of the some\_data.csv file. The delimiter is a comma (,).

```
awk '{sss+= $1} END {print sss}' only_numbers.csv
```

Sums all the numbers in the first column of the only\_numbers.csv file.

More about sed, date and awk: <https://data36.com/bash-more-commands>

## **CREATED BY**

### **Tomi Mester from Data36.com**

Tomi Mester is a data analyst and researcher. He's worked for Prezi, iZettle and several smaller companies as an analyst/consultant. He's the author of the Data36 blog where he writes posts and tutorials on a weekly basis about data science, AB-testing, online research and coding. He's an O'Reilly author and presenter at TEDxYouth, Barcelona E-commerce Summit, Stockholm Analytics Day and more.



## **WHERE TO GO NEXT**

**Find company workshops, online tutorials and online video courses on my website:**

<https://data36.com>

Subscribe to my Newsletter list for useful stuff like this:

<https://data36.com/newsletter>

**Online command line and bash tutorial (free):** [data36.com/bash-tutorial](https://data36.com/bash-tutorial)

**6-week Data Science course:** [data36.com/jds](https://data36.com/jds)