

Bash Lecture 2 – Advanced Bash Commands

★ Bibliography:

<https://www.rigacci.org/docs/biblio/online/sysadmin/toc.htm>

<https://www.tldp.org/LDP/abs/html/>

★ Learning Materials:

<http://www.ee.surrey.ac.uk/Teaching/Unix/>

https://github.com/bertocco/abilita_info_units_2021

Arguments of this lesson



- ★ Bash configuration and user's environment manipulation
- ★ Locating commands ([which](#))
- ★ File information commands ([find](#), [file](#))
- ★ UNIX processes
- ★ Process related commands ([kill](#), [ps](#), [wait](#), [nohup](#), [sleep](#))
- ★ File content related commands ([more](#), [less](#), [tail](#), [read](#), [tee](#), [wc](#))
- ★ Redirection
- ★ File content search commands ([grep](#), [egrep](#))
- ★ Status commands
- ★ Unix wildcards ([*](#) [?](#) [\[\]](#))

Bash configuration



Bash has more configuration startup files.
They are executed at bash start-up time.
The files and sequence of the files executed differ from the type of shell. Shell can be:

- ★ Interactive
- ★ Non-interactive
- ★ Login shell
- ★ Non-login shell

Bash types



- ★ **Interactive**: means that the commands are run with user-interaction from keyboard. E.g. the shell can prompt the user to enter input.
- ★ **Non-interactive**: the shell is probably run from an automated process. Typically input from standard input and output to log file.
- ★ **Login**: shell is run as part of the login of the user to the system.
- ★ **Non-login**: any shell run by the user after logging on, or run by any automated process not coupled to a logged in user.

Bash startup files (1)



http://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

★ **Interactive** login shell, or with **–login**:

/etc/profile, if that file exists

~/.bash_profile,

~/.bash_login

~/.profile, in that order, and reads and executes

--noprofile option may be used to inhibit this behavior.

When an interactive login shell exits, or a non-interactive login shell executes the exit builtin command, Bash reads and executes commands from the file ~/.bash_logout, if it exists.

Bash startup files (2)



http://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

★ **interactive** non-login shell

[Example: when you open a new terminal window by pressing Ctrl+Alt+T, or just open a new terminal tab.]

bash reads and executes commands from

~/.bashrc, if that file exists.

- norc option to inhibit this behaviour.

- rcfile file option will force Bash to read and execute commands from file instead of ~/.bashrc.

So, typically, your ~/.bash_profile contains the line

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

after (or before) any login-specific initializations.

Bash startup files (3)



http://www.gnu.org/software/bash/manual/html_node/Bash-Startup-Files.html

★Invoked non-interactively

[Example: to run a shell script]

bash looks for the variable `BASH_ENV` in the environment, expands its value if it appears there, and uses the expanded value as the name of a file to read and execute. Bash behaves as if the following command were executed:

```
if [ -n "$BASH_ENV" ]; then . "$BASH_ENV"; fi
```

but the value of the `PATH` variable is not used to search for the filename.

If a non-interactive shell is invoked with the `--login` option, Bash attempts to read and execute commands from the login shell startup files.

Set user's PATH environment variable



```
$ cat .profile

# ~/.profile: executed by the command interpreter for login shells.
# This file is not read by bash(1), if ~/.bash_profile or
# ~/.bash_login exists.

# if running bash
if [ -n "$BASH_VERSION" ]; then
    # include .bashrc if it exists
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi

# set PATH so it includes user's private bin directories
PATH="$HOME/bin:$HOME/.local/bin:$PATH"
```

~/.bashrc file



User specific, hidden by default.

~/.bashrc

If not there simply create one.

System wide:

/etc/bash.bashrc

`export`



`export` exports environment variables (also to children of the current process). **Example:**

```
ubuntu~$ export a=test_env
```

```
ubuntu:~$ echo $a
```

```
test_env
```

```
ubuntu:~$ /bin/bash
```

```
ubuntu:~$ echo $a
```

```
test_env
```

```
ubuntu:~$ exit
```

```
exit
```

```
ubuntu:~$ echo $a
```

```
test_env
```

`export` called with no arguments prints all of the variables in the shell's environment.

`unset` frees variables

Shell alias



A shell alias is a shortcut to reference a command. It can be used to avoid typing long commands or as a means to correct incorrect input.

Example: it is used to set default options on commands

```
alias ls='ls -l'
alias rm='rm -i'
```

Exercises:

- 1) try to define and use the previous aliases
- 2) Define the aliases in the `~/.bashrc`, open a new terminal and verify the aliases running them

Locating commands



- ★ To execute a command, UNIX has to locate the command before it can execute it
- ★ UNIX uses the concept of **search path** to locate the commands.
- ★ Search path is a list of directories in the order to be searched for locating commands. Usually it contains standard paths (`/bin`, `/usr/bin`, ...)
- ★ Modify the search path for your environment modifying the `PATH` environment variable

`which`



★ `which` can be used to find whether a particular command exists in your search path. **If it does exist,** which tells you which directory contains that command.

Examples (try with existing and not existing commands):

which pippo

which gedit

which vim

User related commands: passwd



★ `passwd` changes user's password

Example: type `passwd`

```
$ passwd
```

```
Changing password for bertocco.
```

```
(current) UNIX password:
```

```
Enter new UNIX password:
```

```
Retype new UNIX password:
```

```
Sorry, passwords do not match
```

```
passwd: Authentication token manipulation error
```

```
passwd: password unchanged
```

```
$ passwd
```

```
Changing password for bertocco.
```

```
(current) UNIX password:
```

```
Enter new UNIX password:
```

```
Retype new UNIX password:
```

```
passwd: password updated successfully
```

User related commands: who and whoami



★ `who` show who is logged on

Print information about users who are currently logged in.

★ `whoami`

Print the user name associated with the current effective user ID.

Exercise:

try the commands and then type `man who`
and try some option

File information commands



★ Each file and directory in UNIX has several attributes associated with it. UNIX provides several commands to inquire about and process these attributes

`find`



★ **`find`** searches for the particular file giving the flexibility to search for a file by various attributes: name, size, permission, and so on.

Command general form:

find directory-name search-expression

`find` Examples (try)



```
find . -name pippo
```

```
find /etc -name networking
```

```
find /etc -name netw    # nothing found
```

```
find /etc -name netw\*
```

```
find -size 18          # 18 blocks files
```

```
find -size 1024c      # 1024 bytes
```

```
find . -print
```

Read the manual and try other options

Try, if possible, a find case insensitive

★ **`file`** can be used to determine the type of the specified file.

Examples (try):

```
$ file /etc/networking/interfaces
```

```
/etc/networking/interfaces: cannot open
```

```
`/etc/networking/interfaces' (No such file or directory)
```

```
$ file /etc/network/interfaces
```

```
/etc/network/interfaces: ASCII text
```

UNIX Processes



Usually, a command or a script that you can execute consists of one or more processes.

The processes can be categorized into the following broad groups:

- ★ **Interactive processes**, which are those executed at the terminal.

Can execute either in foreground or in background. In a foreground process, the input is accepted from standard input, output is displayed to standard output, and error messages to standard error. In background, the terminal is detached from the process so that it can be used for executing other commands. It is possible to move a process from foreground to background and vice versa (`<ctrl+bg>`; `<ctrl+fg>`).

- ★ **Batch processes** are not submitted from terminals. They are submitted to job queues to be executed sequentially.

- ★ **Deamons** are never-ending processes that wait to service requests from other processes.

Process Related Commands



★ a command or a script that you can execute consists of one or more processes.

The main are:

- ``ps``
- ``kill``
- ``wait``
- ``nohup``
- ``sleep``

`ps`



★ `ps` command is used to find out which processes are currently running.

Exercises:

- Try the following commands, check the differences in the output. Read the flag meaning using

``man ps`:`

`ps`

`ps -ef`

`ps -aux`

- ★ `kill` is used to send signals to an executing process. The process must be a nonforeground process for you to be able to send a signal to it using this command.
- ★ The default action of the command is to terminate the process by sending it a signal. If the process has been programmed for receiving such a signal. In such a case, the process will process the signal as programmed.
- ★ You can kill only the processes initiated by you. However, the root user can kill any process in the system.
- ★ The flags associated with the kill commands are as follows:
 - l to obtain a list of all the signal numbers and their names that are supported by the system.
 - 'signal number' is the signal number to be sent to the process. You can also use a signal name in place of the number. The strongest signal you can send to a process is 9 or kill.

`kill` Exercises



- ★ Look for a process PID of a process belonging of you (using `ps`) and kill it using two different signals: -9 and -15.
- ★ List all available signals and red the differences between the two signal previously used

`wait` with exercises



★ `wait` is to wait for completion of jobs. It takes one or more process IDs as arguments. This is useful while doing shell programming when you want a process to be finished before the next process is invoked.

If you do not specify a process ID, UNIX will find out all the processes running for the current environment and wait for termination of all of them.

★ Examples:

- `wait` If you want to find out whether all the processes you have started have completed
- `wait 15060` If you want to find out whether the process ID 15060 has completed

★ The return code from the wait command is zero if you invoked the wait command without any arguments. If you invoked the wait command with multiple process IDs, the return code depends on the return code from the last process ID specified.

`wait`: exercise



- ★ From a shell launch an infinite process using:
``while true; do echo looping; sleep 2; done``
- ★ From another shell find the pid of this process using
``ps`` command
- ★ From a third shell launch a process waiting for the end of the initial infinite loop
`pid=<your_process_pid>; wait $pid`
- ★ From a fourth shell kill the first process (pid)
- ★ Check in the third shell that your waiting process ended

NOT WORKING using shells. Needs scripting: next time.

`nohup`



★ When you are executing processes under UNIX, they can be running in foreground or background. In a foreground process, you are waiting at the terminal for the process to finish. Under such circumstances, you cannot use the terminal until the process is finished. You can put the foreground process into background as follows:

`ctrl-z`

`bg`

The processes in UNIX will be terminated when you logout of the system or exit the current shell whether they are running in foreground or background. The only way to ensure that the process currently running is not terminated when you exit is to use the `nohup` command.

`nohup`



The `nohup` command has default redirection for the standard output. It redirects the messages to a file called `nohup.out` under the directory from which the command was executed. That is, if you want to execute a script called `sample_script` in background from the current directory, use the following command:

```
nohup sample_script &
```

The `&` (ampersand) tells UNIX to execute the command in background. If you omit the `&`, the command is executed in foreground. In this case, all the messages will be redirected to `nohup.out` under the current directory. If the `nohup.out` file already exists, the output will be appended to it.

`nohup`: Examples



```
nohup grep sample_string * &
```

```
nohup grep sample_string * > mygrep.out &
```

```
nohup my_script > my_script.out &
```

`sleep`



`sleep` wait for a certain period of time between execution of commands. This can be used in cases where you want to check for, say, the presence of a file, every 15 minutes. The argument is specified in seconds.

Examples: If you want to wait for 5 minutes between commands, use:

```
sleep 300
```

Small shell script that reminds you twice to go home, with a 5-minute wait between reminders:

```
echo "Time to go home"
```

```
sleep 300
```

```
echo "Final call to go home ....."
```

File Content Related Commands



★ Commands that can be used to look at the contents of the file or parts of it. You can use these commands to look at the top or bottom of a file, search for strings in the file, and so on.

★ `more` can be used to display the contents of a file one screen at a time. By default, the more command displays one screen worth of data at a time. The more command pauses at the end of display of each page. To continue, press a space bar so that the next page is displayed or press the Return or Enter key to display the next line. Mostly the more command is used where output from other commands are piped into the more command for display.

★ Try

`less`



★ `less` is to view the contents of a file. This may not be available by default on all UNIX systems. It behaves similarly to the more command. The less command allows you to go backward as well as forward in the file by default.

★ Try

★ `Cat <a big file> | less`

★ `tail` to display, on standard output, a file starting from a specified point from the start or bottom of the file. Whether it starts from the top of the file or end of the file depends on the parameter and flags used. One of the flags, -f, can be used to look at the bottom of a file continuously as it grows in size. By default, tail displays the last 10 lines of the file.

`tail` exercises



```
tail -f500 /var/log/syslog
```

list of flags that can be used with the tail command:

- c number to start from the specified character position number.
- b number to start from the specified 512-byte block position number.
- k number to start from the specified 1024-byte block position number.
- n number to start display of the file in the specified line number.
- r number to display lines from the file in reverse order.
- f to display the end of the file continuously as it grows in size.

`read`



`read` is used in shell scripts to read each field from a file and assign them to shell variables.

A field is a string of bytes that are separated by a space or newline character. If the number of fields read is less than the number of variables specified, the rest of the fields are unassigned.

Flag `-r` to treat a `\`(backslash) as part of the input record and not as a control character.

`read` Examples



Example following is a piece of shell script code that reads first name and last name from namefile and prints them:

- create the file

```
cat <<EOF > names_list.txt
```

```
Sara Bertocco
```

```
Mario Rossi
```

```
John Doe
```

```
EOF
```

- Read the file by line and print on standard output

```
while read -r lname fname
```

```
do
```

```
    echo "$lname", "$fname
```

```
done < names_list.txt
```

`read` Examples



Example following is a piece of shell script code that reads a file by line:

```
while read -r line
do
    printf 'Line: %s\n' "$line"
done < names_list.txt
```

The file name can be indicate also with full path name.

`tee`



`tee` to execute a command and want its output redirected to multiple files in addition to the standard output, use the tee command. The tee command accepts input from the standard input, so it is possible to pipe another command to the tee command.

The default of the tee command is to overwrite the specified file.

`-a` is an optional flag to append to the end of the specified file

`tee` Examples (try)



- use the cat command on file1 to display on the screen and make a copy of file1 on file2, use the tee command as follows:

```
cat file1 | tee file2 | more
```

- make the same but appending file1 to the end of an already existing file2 using the flag -a :

```
cat file1 | tee -a file2 | more
```

`wc` counts the number of bytes, words, and lines in specified files. A word is a number of characters stringed together delimited either by a space or a newline character.

Following is a list of flags that can be used with the `wc` command:

- l to count only the number of lines in the file.
- w to count only the number of words in the file.
- c to count only the number of bytes in the file.

You can use multiple filenames as argument to the `wc` command.

`wc` exercises



`wc file`

`wc -w file`

`cat <file> | wc -l`

`wc -w <file1> <file2>`

Redirection (1)



Each UNIX command (or program) is connected to three communication channels between the command and its environment:

- Standard input (stdin) where the command read its input
- Standard output (stdout) where the command writes its output
- Standard error (stderr) where the command writes its error

When a command is executed via an interactive shell, the streams are typically connected to the text terminal on which the shell is running, but can be changed with redirection or with a pipeline

redirect stdout to a file	redirect stderr and stdout to a file
redirect stderr to a file	redirect stderr and stdout to stdout
redirect stdout to stderr	redirect stderr and stdout to stderr
redirect stderr to stdout	

Standard Input, Standard Output and Standard Error Symbols:

standard input	0<
standard output	1>
standard error	2>

Redirection [`>` `&>` `>&` `>>&`].

- Redirect stdout to file (overwrite filename if it already exists):

scriptname > filename

```
scriptname >> filename    # appends the output of scriptname to file filename. If
                           # filename does not already exist, it is created
```

- Redirect stderr to file (overwrite filename if it already exists):

scriptname 2> filename

- Redirect both the stdout and the stderr of command to filename:

command &> filename redirects both the stdout and the stderr of command to filename

- Redirects stdout of command to stderr:

command >&2

- Redirects stderr of command to stdout:

```
command 2>&1
```

Redirection: Examples



- Stdout redirected to file

```
find . -name pippo > find-output.txt
```

- Stderr redirected to file

```
find . -name pippo 2> find-errors.txt
```

- discards any errors that are generated by the find command

```
find / -name "*" -print 2> /dev/null
```

/dev/null is a simple device (implemented in software and not corresponding to any hardware device on the system).

/dev/null looks empty when you read from it.

Writing to /dev/null does nothing: data written to this device simply "disappear."

Often a command's standard output is silenced by redirecting it to /dev/null, and this is perhaps the null device's commonest use in shell scripting:

```
command > /dev/null
```

- Redirect both stdout and stderr to file

```
find . -name pippo &> out_and_err.txt
```

- Redirect stderr to stdout:

```
find . -name filename 2>&1
```

- Redirect stdout to stderr:

```
find . -name filename 1>&2
```

Special characters: Pipe



Pipe [|]. Passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell. This is a method of chaining commands together.

```
echo ls -l | sh
```

Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".

```
cat *.lst | sort | uniq
```

Merges and sorts all ".lst" files, then deletes duplicate lines.

A pipe sends the stdout of one process to the stdin of another. In a typical case, a command, such as cat or echo, pipes a stream of data to a command that transforms it in input for processing:

```
cat $filename1 $filename2 | grep $search_word
```

Redirection with pipe and tee examples



Examples of redirection of the output of a command to be used as input of another:

- Display the output of a command (in this case `ls`) by pages:
`ls -la | less`
- Count files in a directory:
`ls -l | wc -l`
- Count the number of rows containing of the word “canadesi” in the file `vialactea.txt`
`grep canadesi vialactea.txt | wc -l`
- Count the number of words in the rows containing the word “canadesi”

‘tee’ is useful to redirect output both to stdout and to a file. Example:

```
find . -name filename.ext 2>&1 | tee -a log.txt
```

This will take stdout and append it to log file. The stderr will then get converted to stdout which is piped to tee which appends it to the log and sends it to stdout which will either appear on the tty or can be piped to another command.

To go deep: <https://stackoverflow.com/questions/2871233/write-stdout-stderr-to-a-logfile-also-write-stderr-to-screen>

Exercise: redirection



Create a directory and file tree like this one:

```
my_examples /ex1.dir  
            /ex2.txt  
            /ex3.dir  
            /ex3.dir/file1.txt  
            /ex3.dir/file2.txt  
            /ex3.dir/file3.txt
```

Remove read permissions to directory */ex2.dir*

Redirect output on a file. Error is displayed on terminal

Redirect error on a file. Output is displayed on terminal

Verify the content of the files

Stderr redirected to file

Redirect output and errors simultaneously

Use pipe to redirect the output of a command to another command and to a file

Use tee to redirect output both to stdout and to a file

File Content Search Commands



For searching for a pattern in one or more files, use the `grep` series of commands. The `grep` commands search for a string in the specified files and display the output on standard output.

`egrep`



`egrep` extended version of grep command. This command searches for a specified pattern in one or more files and displays the output to standard output. The pattern can be a regular expression to match any single character.

- * to match one or more single characters that precede the asterisk.
- ^ to match the regular expression at the beginning of a line.
- \$ to match the regular expression at the end of a line.
- + to match one or more occurrences of a preceding regular expression.
- ? to match zero or more occurrences of a preceding regular expression.
- [] to match any of the characters specified within the brackets.

`egrep` Examples



Let us assume that we have a file called file1 whose contents are shown below using the more command:

more file1

****** This file is a dummy file ******

which has been created

to run a test for egrep

grep series of commands are used by the following types of people

programmers

end users

Believe it or not, grep series of commands are used by pros and novices alike

****** THIS FILE IS A DUMMY FILE ******

`egrep` Examples



- If you are just interested in finding the number of lines in which the specified pattern occurs, use the `-c` flag as in the following command:
`egrep -i -c dummy file1`
- If you want to get a list of all lines that do not contain the specified pattern, use the `-v` flag as in the following command:
`egrep -i -v dummy file1`
- If you are interested in searching for a pattern that you want to search as a word, use the `-w` flag as in the following command:
`egrep -w grep file1`

`egrep` Examples



- If you want to find all occurrences of dummy, use the following command:

```
egrep dummy file1
```

```
***** This file is a dummy file *****
```

- If you want to find all occurrences of dummy, irrespective of the case, use the -i flag as in the following command:

```
egrep -i dummy file1
```

```
***** This file is a dummy file *****
```

```
***** THIS FILE IS A DUMMY FILE *****
```

- If you want to display the relative line number of the line that contains the pattern being searched, use the -n flag as in the following command:

```
egrep -i -n dummy file1
```

```
1:***** This file is a dummy file *****
```

```
8:***** THIS FILE IS A DUMMY FILE *****
```

★ Several commands that display the status of various parts of the system. These commands can be used to monitor the system status at any point in time.

`date`



`date` command to display the current date and time in a specified format. If you are root user, use the date command to set the system date.

To display the date and time, you must specify a + (plus) sign followed by the format. The format can be as follows:

%A to display date complete with weekday name.

%b or %h to display short month name.

%B to display complete month name.

%c to display default date and time representation.

%d to display the day of the month as a number from 1 through 31.

%D to display the date in mm/dd/yy format.

%H to display the hour as a number from 00 through 23.

%I to display the hour as a number from 00 through 12.

%j to display the day of year as a number from 1 through 366.

%m to display the month as a number from 1 through 12.

%M to display the minutes as a number from 0 through 59.

%p to display AM or PM appropriately.

%r to display 12-hour clock time (01-12) using the AM-PM notation.

%S to display the seconds as a number from 0 through 59.

`date`



Other format flags:

%T to display the time in hh:mm:ss format for 24 hour clock.

%U to display the week number of the year as a number from 1 through 53 counting Sunday as first day of the week.

%w to display the day of the week as a number from 0 through 6 with Sunday counted as 0.

%W to display the week number of the year as a number from 1 through 53 counting Monday as first day of the week.

%x to display the default date format.

%X to display the time format.

%y to display the last two digits of the year from 00 through 99.

%Y to display the year with century as a decimal number.

%Z to display the time-zone name, if available.

`date`: Exercises



Try some example of `date` command usage with different display of day, month, year

- ★ If you want to display the date without formatting, use date without any formatting descriptor as follows:

```
date
```

```
Sat Dec 7 11:50:59 EST 1996
```

- ★ If you want to display only the date in mm/dd/yy format, use the following commands:

```
date +%m/%d/%y  
12/07/96
```

- ★ If you want to format the date in yy/mm/dd format and time in hh:mm:ss format, use the following command:

```
date "+%y/%m/%d %H:%M:%S"  
96/12/07 11:57:27
```

- ★ Following is another way of formatting the date:

```
date +%A,"%B" "%d", "%Y"  
Sunday, December 15, 1996
```

There are three main wildcards in Linux:

Asterisk (*) – matches one or more occurrences of any character, including no character.

Question mark (?) – represents or matches a single occurrence of any character.

Bracketed characters ([]) – matches any occurrence of character enclosed in the square brackets.

Editor

- ★ In dictionary.cambridge.org: is a piece of software for editing text on a computer
- ★ In www.merriam-webster.com: is a computer program that permits the user to create or modify data (such as text or graphics) especially on a display screen

Editor types



In Linux, text editor are of two kinds:

★ graphical user interface (GUI) based

- gedit
- bluefish
- lime

★ command line text editors (console or terminal)

- nano
- pico
- vi/vim
- emacs

Nano



Nano is the built-in basic text editor for many popular linux distros. It doesn't take any learning or getting used to, and all its commands and prompts are displayed at the bottom.

★ Use Nano if:

- You're new to the terminal
- you just need to get into a file for a quick change.

Compared to more advanced editors in the hands of someone who knows what they're doing, some tasks are cumbersome and non-customizable.

★ How to Nano:

from your terminal, enter ``nano`` and the filename you want to edit.

If the file doesn't already exist, it will once you save it.

Commands are listed across the bottom and are triggered with the Control (CTRL) key. For example, to find something in your file, hold CTRL and press W, tell it what you're searching for, and press Enter. Press CTRL+X to exit, then follow the prompts at the bottom of the screen.

GNU emacs



Emacs has so many available features like a terminal, calculator, calendar, email client, web browser, and Tetris, it's often spoken of as an operating system itself.

Starting Emacs is relatively simple, but more you learn, the more there is to learn.

★ How to Emacs:

Emacs commands are accessed through keyboard combinations of CTRL or ALT and another keystroke. When you see shortcuts that read C-h or M-x, C stands for the control key and M stands for the Alt key (or Escape, depending on your system).

Enter ``emacs`` in your terminal, and access the built-in tutorial with C-h t. That means, while holding CTRL, press H, then T.

Or, try key combination C-h r to open the manual within Emacs. You can also use the manual as a playground; just remember to quit without saving by pressing key combination C-x C-c.

Helpful Emacs links



- ★ Emacs wiki

<https://www.emacswiki.org/emacs/SiteMap>

- ★ GNU Guided Tour

<https://www.gnu.org/software/emacs/tour/>

- ★ Cornell Emacs Quick Reference

<https://www.cs.cornell.edu/courses/cs312/2006fa/software/quick-emacs.html>

Main Emacs commands



Principali comandi di emacs	
Undo	CTRL-x u oppure CTRL-_
Salva il file	CTRL-x CTRL-s
Salva con nome diverso	CTRL-x CTRL-w <i>nome</i>
Apri un nuovo file	CTRL-x CTRL-f <i>nome</i>
Inserisce un file	CTRL-x i <i>nome</i>
Passa ad un altro buffer	CTRL-x b
Chiude un buffer	CTRL-x k
Divide la finestra in due	CTRL-x 2
Passa da una metà all'altra	CTRL-x o
Riunifica la finestra	CTRL-x 1
Refresh della finestra	CTRL-l
Quit da emacs	CTRL-x CTRL-c
Cursore a fine riga	CTRL-e
Cursore a inizio riga	CTRL-a
Cursore giù una pagina	CTRL-v
Cursore su una pagina	ESC v
Inizio del buffer	ESC <
Fine del buffer	ESC >
Vai alla linea...	ESC x goto-line <i>numero</i>

Cerca testo	CTRL-s <i>testo</i>
Sostituisce testo	ESC % <i>testo1 testo2</i>
Marca inizio di un blocco	CTRL-SPACE
Marca fine blocco e taglia	CTRL-w
Marca fine blocco e copia	ALT-w
Incolla blocco	CTRL-y
Pagina di aiuto	CTRL-h CTRL-h
Significato di un tasto	CTRL-h k <i>tasto</i>
Significato di tutti i tasti	CTRL-h b
Interrompe comandi complessi	CTRL-g
Apri una shell dentro emacs	ESC x shell
Aiuto psicologico	ESC x doctor
Torri di Hanoi	ESC x hanoi

<http://www.di.unipi.it/~bozzo/fino/appunti/node2.html>

Vi, typically comes with your distro-of-choice.

Vim is a vi successor with some improvements. It runs by default on OS X and some Linux distributions when `vi` is run.

VI has two modes of operation (is a “modal” editor):

- Command mode for navigating files: commands which cause action to be taken on the file
- Insert mode for editing text: in which entered text is inserted into the file.

Because Vi is navigated through the use of keyboard commands and shortcuts, it is better experienced than explained.

How to Vi or Vim



Enter ``vi`` or ``vim`` in your terminal.

When you enter Vi, you begin in command mode and navigate using keyboard commands and the H, J, K, and L keys to move left, down, up, and right, respectively (but arrows use is possible in the most recent versions).

To enter in editing mode press:

`'a'` to append to the file

`'i'` to insert

pressing the `<Esc>` (Escape) key turns off the Insert mode.

To exit Vim without saving, press ESC to enter command mode, then press `:` (colon) to access the command line (a prompt appears at the very bottom) and enter `q!`.

To save and quit, you could use that prompt and the key combination `:wq`, or hold down SHIFT and press Z two times (the shortcut `SHIFT+ZZ`).

The `:` (colon) operator begins many commands like `:help` for help, or `:w` to save.

If you're stuck at the prompt and don't remember the operator you want to use, enter `:` (colon), then press `CTRL+D` for a list of possibilities.

Helpful VI links



★ Basic vi Commands

<https://www.cs.colostate.edu/helpdocs/vi.html>

★ Swathmore's Tips and Tricks

<https://www.cs.swarthmore.edu/oldhelp/vim/home.html>

★ Linux Academy's Vim Reference Guide

<https://linuxacademy.com/blog/wp-content/uploads/2016/06/vim-2.png>

vi basic commands

Summary of most useful commands

©Copyright 2017-2005, Free Electrons, <http://free-electrons.com>. Latest update: Feb 8, 2017
Free to share under the terms of the Creative Commons Attribution-ShareAlike 3.0 license, Electronic copies, sources, translations and updates: <http://free-electrons.com/doc/legacy/command-line/>. Thanks to: Liubo Chen.

Entering command mode

[Esc] Exit editing mode. Keyboard keys now interpreted as commands.

Moving the cursor

h (or left arrow key) move the cursor left.
l (or right arrow key) move the cursor right.
j (or down arrow key) move the cursor down.
k (or up arrow key) move the cursor up.
[Ctrl] f move the cursor one page forward .
[Ctrl] b move the cursor one page backward.
^ move cursor to the first non-white character in the current line.
\$ move the cursor to the end of the current line.
G go to the last line in the file.
nG go to line number *n*.
[Ctrl] G display the name of the current file and the cursor position in it.

Entering editing mode

i insert new text before the cursor.
a append new text after the cursor.
o start to edit a new line after the current one.
O start to edit a new line before the current one.

Replacing characters, lines and words

r replace the current character (does not enter edit mode).
s enter edit mode and substitute the current character by several ones.
cw enter edit mode and change the word after the cursor.
C enter edit mode and change the rest of the line after the cursor.

Copying and pasting

yy copy (yank) the current line to the copy/paste buffer.
p paste the copy/paste buffer after the current line.
P Paste the copy/paste buffer before the current line.

Deleting characters, words and lines

All deleted characters, words and lines are copied to the copy/paste buffer.

x delete the character at the cursor location.

dw delete the current word.
D delete the remainder of the line after the cursor.
dd delete the current line.

Repeating commands

. repeat the last insertion, replacement or delete command.

Looking for strings

/string find the first occurrence of *string* after the cursor.
?string find the first occurrence of *string* before the cursor.
n find the next occurrence in the last search.

Replacing strings

Can also be done manually, searching and replacing once, and then using n (next occurrence) and . (repeat last edit).

n,ps/str1/str2/g between line numbers *n* and *p*, substitute all (*g*: global) occurrences of *str1* by *str2*.
1,\$s/str1/str2/g in the whole file (\$: last line), substitute all occurrences of *str1* by *str2*.

Applying a command several times - Examples

5j move the cursor 5 lines down.
30dd delete 30 lines.
4cw change 4 words from the cursor.
1G go to the first line in the file.

Misc

[Ctrl] l redraw the screen.
J join the current line with the next one
u undo the last action

Exiting and saving

ZZ save current file and exit vi.
:w write (save) to the current file.
:w file write (save) to the *file* file.
:q! quit vi without saving changes.

Going further

vi has much more flexibility and many more commands for power users!

It can make you extremely productive in editing and creating text.

Learn more by taking the quick tutorial: just type **vimtutor**.



The editor war



Technologies available in Information Technology are a lot. Often, to solve a problem, you can choose between different instruments. The rule to base your choose is: It does not exist “the best tool” but “**the best tool to solve your specific problem**”.

Sometimes different tools are more or less equivalent.

This is the case of editors emacs and vim:

https://en.wikipedia.org/wiki/Editor_war

Choose your editor



Try an editor and its tutorial,
watch videos on how to use it for your intended purpose,
spend a day or two using it with real files training your fingers.

The best editor for you is the one that makes you feel like
you're easily getting things done.