Bash Lecture 3 – Bash Scripting

Bibliography and learning materials

★ 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

- ★Process related commands (nohup, sleep)
- ★File content related commands (more, less, tail, read, tee, wc)
- ★Redirection
- ★File content search commands (grep, egrep)
- ★Status commands
- ★Unix wildcards (* ? [ ])
- ★Editors
- ★Bash scripting programming:
  - Scripts
  - Variables

`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`

- ★`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`

★`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 indicated 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. 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`

`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` exercices

```
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 (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 symultaneously

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 *****
```

Status commands

★ 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
```

Linux wildcards

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.

Editors

Editor

★In dictionary.cambridge.org: is a piece of software for editing text on a compute

★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.

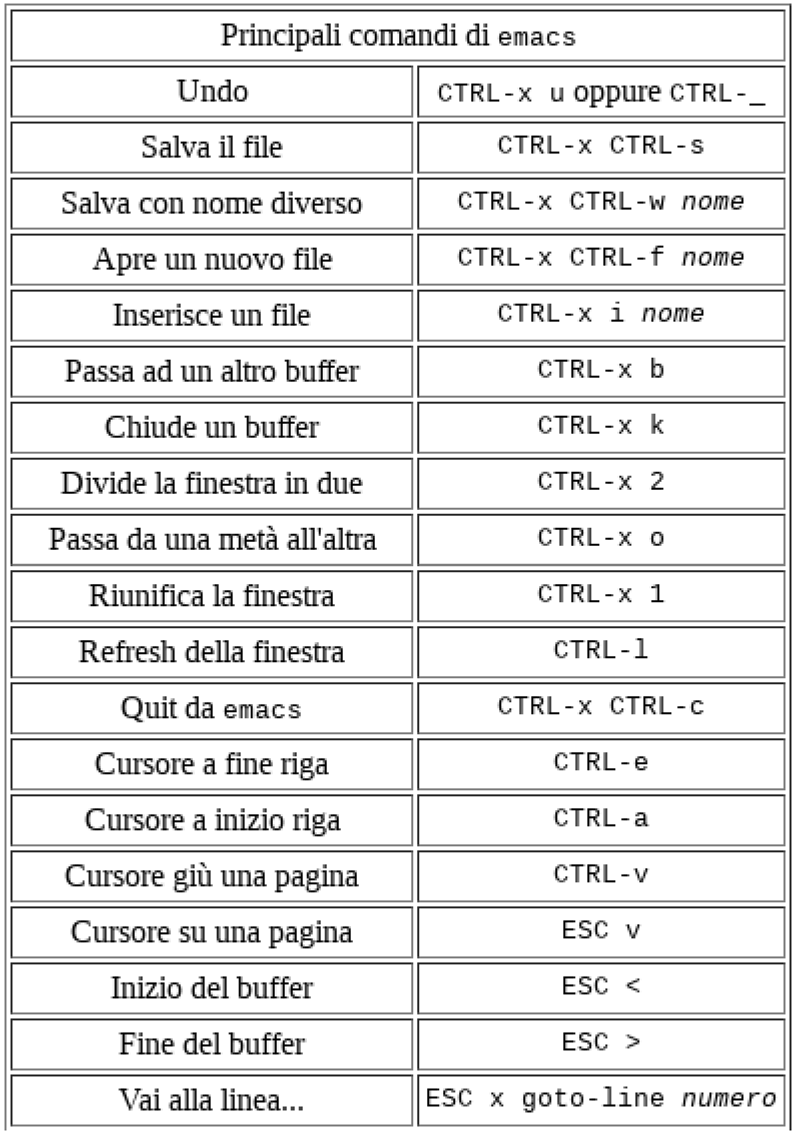
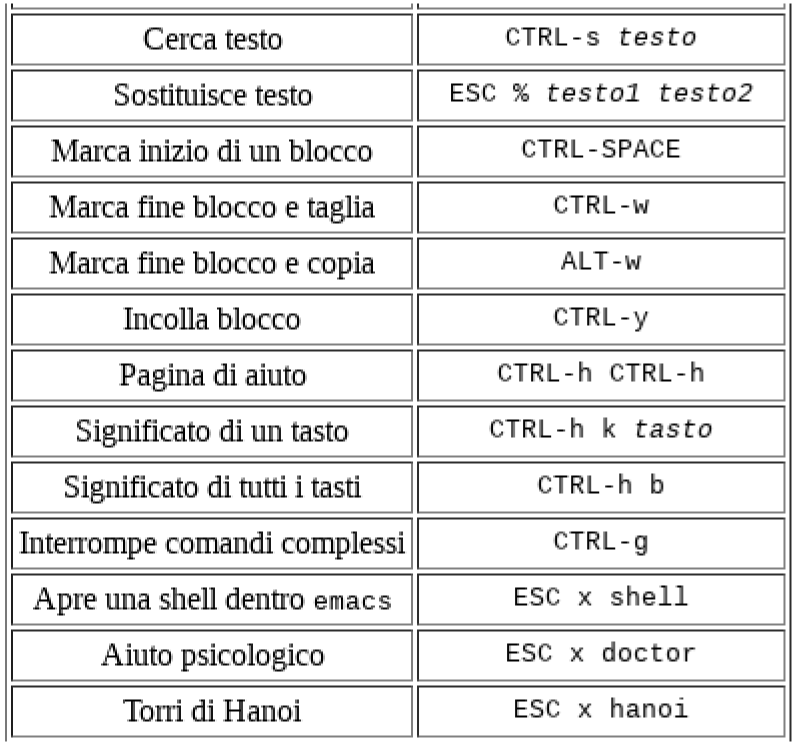
Helpful Emacs links

- ★ 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` *nome*
Apre un nuovo file	`CTRL-x CTRL-f` *nome*
Inserisce un file	`CTRL-x i` *nome*
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` *numero*
Cerca testo	`CTRL-s` *testo*
Sostituisce testo	`ESC %` *testo1 testo2*
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` *tasto*
Significato di tutti i tasti	`CTRL-h b`
Interrompe comandi complessi	`CTRL-g`
Apre 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/vim

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.

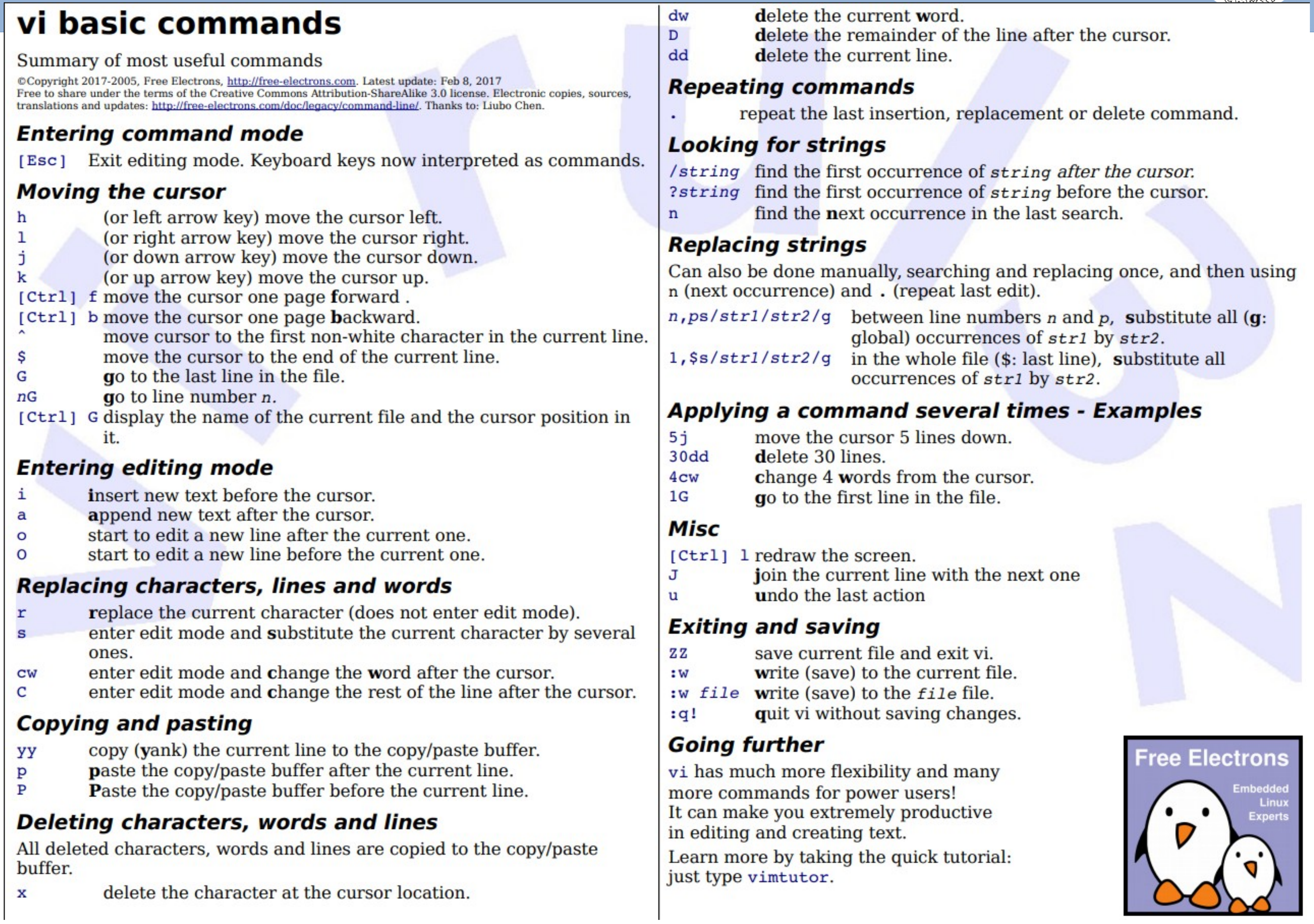
If you’re stuck at the prompt and don’t remember the operator you want to use, enter : (colon), then press CTRL+D for a list of possibilities.

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://www.linuxtrainingacademy.com/vim-cheat-sheet/
https://acloudguru.com/blog/engineering/a-vim-cheat-sheet-reference-guide



vi basic commands

Summary of most useful commands

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 **f**orward .
`[Ctrl] b`	move the cursor one page **b**ackward.
`^`	move cursor to the first non-white character in the current line.
`$`	move the cursor to the end of the current line.
`G`	**g**o to the last line in the file.
`nG`	**g**o to line number *n*.
`[Ctrl] G`	display the name of the current file and the cursor position in it.

Entering editing mode

`i`	**i**nsert new text before the cursor.
`a`	**a**ppend 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`	**r**eplace the current character (does not enter edit mode).
`s`	enter edit mode and **s**ubstitute the current character by several ones.
`cw`	enter edit mode and **c**hange the **w**ord after the cursor.
`C`	enter edit mode and **c**hange the rest of the line after the cursor.

Copying and pasting

`yy`	copy (**y**ank) the current line to the copy/paste buffer.
`p`	**p**aste the copy/paste buffer after the current line.
`P`	**P**aste 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`	**d**elete the current **w**ord.
`D`	**d**elete the remainder of the line after the cursor.
`dd`	**d**elete 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 **n**ext 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*, **s**ubstitute all (**g**: global) occurrences of *str1* by *str2*.
`1,$s/str1/str2/g`	in the whole file ($: last line), **s**ubstitute all occurrences of *str1* by *str2*.

Applying a command several times - Examples

`5j`	move the cursor 5 lines down.
`30dd`	**d**elete 30 lines.
`4cw`	**c**hange 4 **w**ords from the cursor.
`1G`	**g**o to the first line in the file.

Misc

`[Ctrl] l`	redraw the screen.
`J`	**j**oin the current line with the next one
`u`	**u**ndo the last action

Exiting and saving

`ZZ`	save current file and exit vi.
`:w`	**w**rite (save) to the current file.
`:w file`	**w**rite (save) to the *file* file.
`:q!`	**q**uit 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`.

Free Electrons
Embedded Linux Experts

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.

Shell scripting abilities

Many shells have scripting abilities:

Executes sequentially multiple commands written in a script as if they were typed from the keyboard.

Most shells offer additional programming constructs that extend the scripting feature into a programming language.

What is a script

A script is, in the simplest case, a list of system commands stored in a file.

Place commands in a script is useful

- to avoid having to retype them time and again
- to be able to modify and customize the script for a particular application
- to use the script as a program/command

The sha-bang #!

Every script starts with the sha-bang (#!) at the head, followed by the full path name of an interpreter.

Examples:

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
```

This tells your system that the file is a set of commands to be fed to the command interpreter indicated by the path.

The command interpreter executes the commands in the script, starting at the top (the line following the sha-bang line), and ignoring comments.

Execute the script

★ The script execution requires the script has “execute” permissions:

chmod +rx scriptname (gives everyone read/execute permission)

chmod u+rx scriptname (gives only the script owner read/execute permission)

★ The script can be executed issuing:

./scriptname

★ The script can be made available as a command:

- moving the script to /usr/local/bin (as root), making it available to all users as a system wide executable. The script could then be invoked by simply typing scriptname [ENTER] from the command-line.
- Including the directory containing the script in the user's $PATH

Exercise: a first script

- ★ Write a script that upon invocation
  1) Says “Hello!”
  2) shows the time and date
  3) The script then saves this information to a logfile
- ★ Make the script executable
- ★ Execute the script
- ★ Make the script available as a command

Special characters (1)

★ Special characters have a meaning beyond its literal meaning

Comments [#]. Lines beginning with a # (with the exception of #!)

```
# This line is a comment.
```

Comments may also occur following the end of a command.

```
echo "A comment will follow." # Comment here.
```

Comments may also follow whitespace at the beginning of a line.

```
    # Note
```

Command separator [semicolon ;] Permits putting two or more commands on the same line.

```
echo hello; echo world
```

Escape [backslash \] This is a mechanism to express litterally a special charactrer.
For example the \ may be used to escape " and ' echoing a string:

```
echo This is a double quote \”  # This is a double quote ”
```

Special characters (2)

Command substitution [backquotes or backticks `]. The `command` construct makes available the output of command for assignment to a variable.

```
a=`pwd`
echo $a   # display the path of your location
```

Wild card [asterisk *]. The * character serves as a "wild card", it matches every filename in a given directory or every character in a string.

Run job in background [and &]. A command followed by an & will run in the background.

```
    bash$ sleep 10 &
    [1] 850
    [1]+  Done                    sleep 10
```

Within a script, commands and even loops may run in the background.
To bring the script in foreground type `fg` or `CTRL Z fg`
To bring the script in background type `fg` or `CTRL Z bg`

Complete reference:
https://www.tldp.org/LDP/abs/html/special-chars.html

Exercise: special characters

- Write a commented command and execute it
- Write two commands on the same row and execute them
- Make the echo of a string containing one or more escaped characters
- Make the echo of a command (like ls or pwd) output
- Use wildcard to list all files starting with ‘a’ in your directory
- Download from github the script infinite_loop_noout.sh, make it executable if needed, execute it in background, recall it in foreground, stop it

UNIX Variables

- ★ Variables are how programming and scripting languages represent data. A variable is a label, a name assigned to a location holding data.
- ★ Standard UNIX variables are split into two categories:
  - environment variables:
    if set at login, are valid for the duration of the session
  - shell variables:
    apply only to the current instance of the shell and are used to set short-term working conditions;

By convention, environment variables have UPPER CASE and shell variables have lower case names.

- ★ Environment variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for variables and if found, will use the values stored.
- ★ Variables can be set: by the system, by you, by the shell, by any program that loads another program.

bash variables

Variable in bash are untyped.

- ★ Bash variables are character strings: can contain a number, a character, a string of characters.
- ★ Depending on context (i.e. depending whether the value of a variable contains only digits or not), bash permits arithmetic operations and comparisons on variables.
- ★ There is no need to declare a variable, just assigning a value to its reference will create it.

bash variables: assignment (1)

It must distinguish between the name (right value) of a variable and its value (left value).

If variable1 is the name of a variable,
then $variable1 is a reference to its value, i.e. the data item it contains.

$variable1 is actually a simplified form of ${variable1}. In contexts where the $variable syntax causes an error, the longer form ${variable} may work.

Referencing (retrieving) the variable value is called variable substitution.

=> No space permitted on either side of = sign when initializing variables.

Example:

```
a=375      # Initialize variable
hello=$a   # No space permitted on either side of = sign when initializing variables.
#     ^ ^
# What happens if there is a space? Bash will treat the variable name as a program to
# execute, and the = as its first parameter. TRY
#
echo hello        # hello ## Not a variable reference, just the string "hello" ...
echo $hello       # 375 ##  This *is* a variable reference, i.e. shows the value.
echo ${hello}   # 375 ##  Likewise a variable reference, as above.
```

assignment disambiguation with {}

In the previous slide: “In contexts where the $variable syntax causes an error, the longer form ${variable} may work”. This is called variable disambiguation.

Example:
If the variable $type contains a singular noun and we want to transform it on a plural one adding an ‘s’, we can't simply add an ‘s’ character to $type since that would turn it into a different variable, $types.
Although we could utilize code contortions such as
echo "Found 42 "$type"s"

the best way to solve this problem is to use curly braces:
echo "Found 42 ${type}s",
which allows us to tell bash where the name of a variable starts and ends

Exercise: bash variables

Try:

1) STR='Hello World!'
```
echo $STR
```

2) Try assignment and echo the variable content:

```
a=5324
a=(1, 3, 4, 6, 5, "otto")        # array
```

3) Very simple backup script example:

```
OF=/tmp/my-backup-$(date +%Y%m%d).tgz
        tar -czf $OF ./subdir_of_where_i_am
```

bash variables: assignment examples

“naked variable”, i.e. lacking ‘$’ in front, is when a variable is being assigned, rather than referenced.

```
# Assignment simple
a=879 ;  echo "The value of \"a\" is $a."

# Assignment using 'let' (arithmetic expression)
let a=16+5;  echo "The value of \"a\" is now $a."

# In a 'for' loop (see for details later in this lesson):
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done

# In a 'read' statement (also a type of assignment):
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."
```

bash variables: assignment examples(2)

```
#!/bin/bash
# With command substitution

a=`echo Hello!`   # Assigns result of 'echo' command to 'a' ...
echo $a

a=`ls -l`         # Assigns result of 'ls -l' command to 'a'
echo $a             # Unquoted, however, it removes tabs and newlines.

echo "$a"         # The quoted variable preserves whitespace.
```

Exercise 3: practice with variables assignment

Try different variable assignments and print the variable content to standard output

- Simple assignment
- Command output assignment

bash variables: quoting

Quoting means just that, bracketing a string in quotes.
This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a wild card character in Regular Expressions).

Partial quoting consists in enclosing a referenced value in double quotes (" ... "). This does not interfere with variable substitution. Sometimes referred also as "weak quoting."

Full quoting consists in using single quotes ('...').
It causes the variable name to be used literally, and no substitution will take place.

Examples (Try):

```
a=352
echo $a    # 352
echo “$a”  # 352
echo ‘$a’  # $a
```

=> Quoting a variable preserves white spaces.

Exercise 2: variables assignment and quoting

In a bash script:

- Assign a variable
- Print the variable value
- Print a string containing the variable value
- Print a string containing the partial quoted variable
- Print the same string fully quoted
- Assign a variable containing multiple spaces
- Print this new variable
- Print this new variable quoted

- Run the script
- Run the script redirecting the output on a file