A short introduction to the UNIX commandline

Jean-Baka Domelevo Entfellner

BecA-ILRI Hub, Nairobi, Kenya

3rd EANBiT residential training
online, 20 July 2020

Basic structure of a commandline

All commandlines look like this:

`<command> <options> <arguments>`

1. command (**compulsory**): either an executable or a builtin shell command. Examples: `ls`, `rm`, `pwd`, `cd`, `export` (try `which cd; which export`)
2. options (optional): either short one-letter form, collapsable (`ls -alth`) or long format (`grep --file=patterns.txt`)
3. arguments (optional, depending on which command): the "main stuff" on which the command operates (`cp source dest`)

My first commands

- `pwd` to **p**rint the current **w**orking **d**irectory
- `cd Dir1` : to **c**hange **d**irectory into `Dir1` (must be present in current dir)
- `cd /var/scratch` : to **c**hange **d**irectory into `/var/scratch` (absolute path)
- `ls` to **lis**t the contents of the current directory
- `ls Dir1` to **lis**t the contents of `Dir1` (must be present in current dir)
- `man ls` to read the **man**ual page about `ls`
- `chmod +x myscript.sh` to make the script `myscript.sh` executable

More commands

- `rm file1` : to **rem**ove `file1` (must be present in current dir)
- `mkdir Dir1` : to create (**m**a**k**e) an empty **dir**ectory within the current directory
- `rmdir Dir1` : to **rem**ove the **dir**ectory `Dir1` (must be present in current dir)
- `mkdir -p Dir1/SubDir/SubSubDir` to create a directory and all its required parents, as necessary (silent command)
- `touch newfile` to create an empty file in the current directory
- `vim newfile` to edit it with my favorite editor

Autocompletion: the **tab** key is your friend

Most important advice #1

Always autocomplete your command line with the **tab** key!

The advantages are many:

1. save typing time
2. avoid mistyping
3. check in real time that you are “on the right track” (e.g. not trying to access folders that don’t exist)

Using the history

Most important advice #2

Browse your command history using the `↑` (up arrow) key!

More tricks with the history:

- see it with `history`
- start a commandline with a space **not** to record it in the history
- `Ctrl+R` to browse it interactively
- use left or right arrow keys to edit the selected command
- `!p` (or `!f`, etc) to re-run the last command starting with `p` (resp. `f`)

Bash character expansion

- a standalone `*` gets expanded into the list of all files and folders in the current directory (see how `ls *` differs from `ls` when working dir contains folders)
- `*` within a string expands to all possible completions of that string, e.g. `ls *.fasta`
- `?` globs one character exactly, e.g. `b?sh` will match `bash` and `bush`, but not `bsh`
- `[]` to provide a list of characters to pick from: `ls file[189]` will pick `file1`, `file8` and `file9` only
- `[]` can also include a range to pick from: `ls file[5-9]` will pick `file5`, `file6`, `file7`, `file8` and `file9`

ls and the details of file permissions

`ls -l` gives a long listing:

```
$ ls -l
total 336
-rw-r--r-- 1 jbde jbde   1776 Jul  2 03:21 bash_intro.aux
-rw-r--r-- 1 jbde jbde  51179 Jul  2 03:21 bash_intro.log
-rw-r--r-- 1 jbde jbde    747 Jul  2 03:21 bash_intro.nav
-rw-r--r-- 1 jbde jbde      0 Jul  2 03:21 bash_intro.out
-rw-r--r-- 1 jbde jbde 249549 Jul  2 03:21 bash_intro.pdf
-rw-r--r-- 1 jbde jbde      0 Jul  2 03:21 bash_intro.snm
-rwxr-xr-x 1 jbde jbde   4424 Jul  2 03:22 bash_intro.tex
-rw-r--r-- 1 jbde jbde     23 Jul  2 03:21 bash_intro.toc
-rw-r--r-- 1 jbde jbde    689 Jul  2 03:21 bash_intro.vrb
-rwxr-xr-x 1 jbde jbde     22 Jul  1 20:10 echo_v.sh
-rw-r--r-- 1 jbde jbde     53 Jul  1 20:31 test_less_than.
-rwxr-xr-x 1 jbde jbde     28 Jul  1 19:38 test_script.sh
-rw-r--r-- 1 jbde jbde    212 Jul  1 19:06 tmp
```

Rights, aka permissions

On normal files:

- `r` to **r**ead (value=4)
- `w` to **w**rite (value=2)
- `x` to e**x**ecute, e.g. to use it as a command (value=1)

On directories:

- `r` to **r**ead the contents of the directory (e.g. to `ls` it or to autocomplete filenames in it)
- `w` to **w**rite (meaning: to create and delete files in it)
- `x` to **traverse** it (i.e. to browse to subfolders)

To whom do those rights apply:

- `u` for the owner of the file or directory (**u**ser)
- `g` for the **g**roup the file or directory belongs to
- `o` for the rest of the world (the "**o**thers")

Changing owner/permissions

- `chown caleb myfile1 myfile2` : give ownership of these files to user `caleb`
- `chgrp team1 myfile1 myfile2` : set group to `team1`
- `chmod 755 file1` : change permissions to `rwxr-xr-x`
- `chmod 744 file1` : change permissions to `rwxr--r--`
- `chmod 400 file1` : change permissions to `r--------`
- `chmod -w file1` : remove “write” right to all
- `chmod o-w file1` : remove “write” for the “rest of the world”
- `chmod u+x,go-w file1` : add “execute” write to user, and remove “write” right for all other users

Redirections

1. redirecting standard output only: `echo "hello" > myfile`
2. redirecting without overwritting, but appending to existing content: `echo "hello" >> myfile`
3. redirecting standard error stream only: `expr 3 / 0 2> errors.txt`
4. redirecting both: `cat /var/log/*.log &> outfile`
5. feeding standard input from a file: `grep abc < file_in`

   same as `cat file_in | grep abc`

Every single process (including your shell) has a standard input stream (code 0), a standard output stream (code 1), and a standard error stream (code 2): try `file /proc/$$/fd/0`

Control flow: **if** . . . **else** constructs

```
if [ -e hello.txt ]
then
  echo "The file exists!"
else
  echo "The file doesn't exist!"
fi
```

Pay careful attention: put spaces after `[` and before `]`!

Same loop as above, but in a one-liner:

```
if [ -e hello.txt ]; then echo "ok"; else echo "no"; fi
```

Control flow: **for** loops

```
for file in *.sh
do
  echo "File ${file} has $(wc -l < ${file}) lines"
done
```

After the `in` keyword must appear some string that will be interpreted as a sequence of tokens separated by spaces, for instance `{0..4}` will be translated into “0 1 2 3 4”.

Same loop as above, but in a one-liner:

```
for file in *.sh; do echo "File ${file} has \
      $(wc -l < ${file}) lines"; done
```

Bash variables: built-ins

To use the value of a shell variable, use the `$` sign before the variable name. A few **built-in** variables:

- `$?` last return value
- `$PWD` the current working directory
- `$SHELL` the shell you're using
- `$#` is the number of commandline arguments (in a script)
- `$*` all the commandline arguments (as a single string)
- `$0` the zero-th positional argument (i.e. the command)
- `$1`, `$2`, . . . the following positional arguments (separated on the commandline by one or more spaces)

Create your own variable names

Beware of spaces when assigning variables!

NO SPACES before or after that equal sign!!

```
myvar=5
```

```
mypath=/var/scratch/jb
```

New variables are created *locally* in the current environment: use `export` to make them persistent.

Try: `z=4 ; bash -c 'echo $z'` vs

```
export z=4 ; bash -c 'echo $z'
```

By default, Bash variables are **strings**:

```
u=4 ; v=20 ; if [ $u \< $v ]; then echo "yes"; fi
```

Working with variables

Variable names MUST NOT start with a digit or a non-letter sign.
Beware where Bash thinks your variable name ends:

```
myvar=1; echo $myvar_2
```

Working with variables

Variable names MUST NOT start with a digit or a non-letter sign.
Beware where Bash thinks your variable name ends:

```
myvar=1; echo $myvar_2
```

Correct syntax: `myvar=1; echo ${myvar}_2`

Bash quoting

Weak quoting with double quotes will not prevent variable interpretation: `a=5; echo "$a"` prints “5”
Quotes are essential to include spaces in your text:
`myvar="hello boy!"`

Strong quoting prevents interpretation of basically everything:
`a=5; echo '$a'` prints “$a”

Command substitution

The purpose of **command substitution** is to execute a command (possibly with calculated arguments) and to store its output in a Bash variable.

Syntax: `$(ls -1 | wc -l)` or `` `ls -1 | wc -l` ``

Example of use: `numfiles=$(ls | wc -l)`

String manipulation with Bash

The construct with curly braces allow elaborate string manipulation:

- `mystring="hello aloha36"; echo ${mystring}` : this you know...
- `${#mystring}` to get the number of characters in the string
- `${mystring%[0-9]*}` deletes **shortest** match from **end** of string
- `${mystring%%[0-9]*}` deletes **longest** match from **end** of string
- `${mystring#*a}` deletes **shortest** match from **beginning** of string
- `${mystring##*a}` deletes **longest** match from **beginning** of string