Lecture 9: Real-life examples

Last update: 20200624

Table of Contents

1. Command history search

When working directly in the terminal we want to speed up the command input as much as possible. Besides, frequently we want to be able to re-use the certain command input again, without re-typing it from scratch. A lot of typing is saved by using `TAB`, which autocompletes the command input to existing commands names (here commands are meant in the broader sense and include also **Bash** functions, aliases, etc.). In the case command is expecting as argument a file or directory, `TAB` will also autocomplete file or directory name, after we have typed in the first few characters and hit `TAB`. Last but not least, `TAB` also autocompletes the **Bash** variable names when we are referencing their content. This is illustrated with the following three simple examples:

```
dirn + TAB
# autocompletes to command 'dirname'

cat fileW + TAB
# autocompletes to file named 'fileWithLengthyName.log'

LongNameVariable=44
echo $Lo + TAB
# autocompletes to 'echo $LongNameVariable'
```

In general, when there are multiple matches for the initial pattern, after pressing once the `TAB` nothing happens, however pressing two times `TAB TAB` lists all matches:

```
ls Lecture_9 + TAB + TAB
# prints Lecture_9.md Lecture_9.html Lecture_9.pdf
```

The precedence of text completion via `TAB` can be summarized as follows: commands and functions first, then files and directories. If we want to alter these default precedence rules in some special cases of interest, we can use the following three special characters:

- if the text is preceded with `$` : variable completion with `TAB` takes precedence
- if the text is preceded with `~` : username completion with `TAB` takes precedence

The autocompletion via `TAB` is a very neat feature and speeds up a lot the typing, but it cannot help us to re-use what we have already typed.

To achieve that, we need to use **Bash** built-in command **history**. After we type in the terminal

```
history
```

all the command input which we have typed in the terminal recently (not necessarily only in the current terminal!) will be printed and enumerated by **Bash**. For instance, the output could be:

```
  525  ls
  526  nano readMe.txt
  527  ls
  528  tar -czf Lecture_8.tar.gz Lecture_8/
  529  ls
  530  cd ..
  531  ls
  532  typora Homework_7.md &
  533  cd ../Lecture_9
```

From where **Bash** has retrieved this detailed information of what we typed of late in the terminal? All previously typed commands are stored by default in the file to which the environment variable **HISTFILE** is pointing to:

```
echo $HISTFILE
```

The printout could look like:

```
/home/abilandz/.bash_history
```

That means that, by default, the history of all our command input is saved in the file `.bash_history` placed in the home directory. By default, at maximum 1000 lines of command input are kept in this file, but that can be changed by modifying the **Bash** environment variable **HISTSIZE**. If we now have a look at the content of the **Bash** command history file:

```
cat /home/abilandz/.bash_history
```

we see that we get a similar printout like the one from the command **history** showed above. The printout is similar, but not the same, and we will now clarify this difference, which sometimes leads to big confusion.

Each time we start a new terminal, the file `~/.bash_history` is read. From that point onward, each terminal maintains its own history (i.e. its own list of all commands we have typed in the terminal). When we exit the terminal, **Bash** updates the `~/.bash_history` file with the history which corresponds to that terminal. Therefore, and very importantly, the current content of `~/.bash_history` will correspond to the last terminal we have closed.

Some frequently used flags for the command **history** and their meanings are summarized here:

- `-c` : clears the history list (but it doesn't clean the content of `~/.bash_history` instantly, remember that this file gets updated automatically only after we exit the terminal!)
- `-d someNumber` : clears the **history** entry only at the line 'someNumber'
- `-a` : forces appending history lines from the current terminal to the history file `~/.bash_history`. With this option, we save permanently all commands we have typed in the history file, even without exiting the terminal

After understanding the **history** mechanism, we now demonstrate how we can directly extract only the entry we need with a few convenient shortcuts:

- Use up and down arrow (or equivalently `Ctrl+N` and `Ctrl+P`) in the terminal to browse through (in the specified order!):
  - terminal's own history
  - the content of `~/.bash_history`
- `Ctrl+R` : inverse history search. After we press `Ctrl+R`, the following prompt appears:

  ```
  (reverse-i-search)`':
  ```

  Now we can type the pattern which will be used to search for some previously used command input that contained that pattern. We can keep pressing `Ctrl+R`, until the command input we are looking for appears. By pressing the right arrow, that command input is copied in the terminal, and we can now reuse it again.

Example: The inverse history search is an extremely handy feature, and we now illustrate it with the concrete example. Imagine a scenario in which we have typed in the terminal for** loop, followed by a lot of other commands:

```
for i in {1..10}; do echo $i; done
... one zillion other commands ...
```

Do we need to retype the whole **for** loop from scratch, in case we need that command input again? It suffices only to do the following:

```
Ctrl+R
(reverse-i-search)`': # type the pattern'fo'
(reverse-i-search)`fo': for i in {1..10}; do echo $i; done
```

Press the right arrow, and the offered result from the inverse history search is copied in the terminal, and can be reused. If the offered result from the inverse history search is not what we need, we can keep pressing `Ctrl+R` to browse through all results which match the specified pattern.

We indicate now how we can directly re-execute any command input from the history list. For instance, if the command

```
history
```

has produced the following output

```
  188  ls
  189  grep Ctrl Lecture_?/Lect*
  190  for i in {1..10}; do echo $i; done
  191  ls
  192  pwd
```

we can execute any command from above just by typing `!commandNumberInTheList`. Given the above output, the following input in the terminal:

```
!190
```

gives immediately

```
for i in {1..10}; do echo $i; done
1
2
3
4
5
6
7
8
9
10
```

For more elaborate cases of retrieving and even editing the command input from **history**, please see the documentation of the command **fc** ('fix command').

Finally, we remark that programmatically we can retrieve the last argument of the previously executed command. This functionality is achieved via the special Bash variable `$_`. If the previously executed command has only one argument, then the content of `$_` is that argument. For instance:

```
mkdir Dir1 Dir2 Dir3
echo $_
# prints Dir3
```

A frequent use case is the following example:

```
ls file_{1..99}.log
rm $_
```

If the first line has expanded in the list of files we want to delete, we can reuse the same brace expansion n the second line as the argument for **rm** command.

Example: How to make a few directories, and automatically move into the last one created?

```
mkdir Dir1 Dir2 Dir3 && cd $_
```

2. Searching for files and directories: find

We have already seen how we can list the content of the specified directory with known location in the filesystem with **ls** command. However, in case we need to search for specific files or directories at unknown location in the filesystem hierarchy, **ls** command cannot be used. Instead, we can use the **Linux** command **find** is which was designed precisely for that sake. This powerful command can perform search by name, by creation, accession and modification date, by owner and permissions etc. In addition, **find** can immediately perform some action on the result of its search (for instance, it can immediately delete all files it has found, rename all directories, etc.).

The generic usage of command **find** can be described as follows:

```
find Where What Action
```

When interpreting its arguments, **find** defaults the first arguments without `-` or `--` as a list of directories in which the search will be performed. Therefore, in the above generic syntax 'Where' stands for one or more directories. After that, **find** expects one or more options starting with `-` or `--` , which will typically nail down what it needs to search for ('What' in the above syntax). Finally, there exists a special option **-exec** after which we can optionally set the commands which **find** will execute immediately on the results it has found ('Action').

The usage of **find** is best illustrated on concrete examples. Let us start with a directory named 'Examples' in which we have the following situation:

```
$ ls Examples/
Directory_0  Directory_1  Directory_2  Directory_3  file_0.dat  file_0.pdf
file_0.png  file_1.dat  file_1.pdf  file_1.png
```

Example 1: Find and print on the screen only the files in the specified directory.

```
find Examples/ -type f
```

The result is:

```
Examples/file_0.png
Examples/file_1.dat
Examples/file_1.pdf
Examples/file_1.png
Examples/file_0.dat
Examples/file_0.pdf
```

By default, the result of **find** is not sorted, but you can trivially, and in general, sort the output of some command by piping to the command named **sort**:

```
find Examples/ -type f | sort
```

The output is now sorted:

```
Examples/file_0.dat
Examples/file_0.pdf
Examples/file_0.png
Examples/file_1.dat
Examples/file_1.pdf
Examples/file_1.png
```

Example 2: Find all subdirectories in the specified directory.

```
find Examples/ -type d
```

The result is:

```
Examples/Directory_3
Examples/Directory_2
Examples/Directory_0
Examples/Directory_1
```

Example 3: Find all files with an extension '.pdf' in the specified directory

```
find Examples/ -type f -name "*.pdf"
```

The result is:

```
Examples/file_1.pdf
Examples/file_0.pdf
```

Example 4: Find all files with an extension '.pdf' and pattern 'file_0' in their name, in the specified directory.

```
find Examples/ -type f -name "*.pdf" -a -name "*file_0*"
```

The result is:

```
Examples/file_0.pdf
```

From the above example, we see that **find** interprets the flag `-a` as the logical `AND`. Similarly, the flag `-o` can be used within **find** as the logical `OR`.

Since the flag `-name` is very frequently used, it deserves some additional clarification. The usage of quotes in the pattern, as in `"*.pdf"`, was essential, because now the special characters will be supplied as the special characters to the **find** command, and will prevent **Bash** to expand them. Dropping quotes round the pattern is a typical mistake when **find** is used:

```
find Examples/ -type f -name *.pdf # WRONG!!
```

The above syntax is wrong, because **Bash** now will first expand the pattern `*.pdf` to match all files in the current working directory (not in the directory `Examples` !) that end with `.pdf`, and only then those fully expanded file names will be supplied to the command **find**. Clearly, this will work only by accident if in the current working directory where we have executed the command **find** there was no a single file which ends in `.pdf`, and therefore `*.pdf` remained unexpanded.

Alternatively, the special symbols can be supplied to **find** with the escaping mechanism `\`. Summarizing everything:

```
find Examples/ -type f -name "*.pdf" # CORRECT
find Examples/ -type f -name '*.pdf' # CORRECT
find Examples/ -type f -name \*.pdf # CORRECT
find Examples/ -type f -name *.pdf # WRONG!!
```

Example 5: Find all files with an extension '.pdf' larger than 10 KB in the specified directory.

```
find pathToDirectory(-ies) -type f -name "*.pdf" -size +10k
```

Here prefix `+` is not trivial, if we would omit it, the flag `-size 10k` would filter out instead the files whose size is exactly 10 KB. Unfortunately, syntax for KB in **find** is a small 'k', and not capital 'K' (like in **ls -lh**), which frequently leads to confusion. Analogously, files which are smaller than 10 KB in size, we would filter out by using prefix `-`, i.e. `-size -10k`.

Example 6: Find all files with an extension '.tex' modified within last 10 days.

```
find pathToDirectory(-ies) -type f -name "*.tex" -mtime -10
```

Similarly as with the option `-size`, the option `-mtime +10` means more than 10 days ago, `-mtime 10` exactly 10 days ago, and `-mtime -10` less than 10 days ago. Closely related flags are `-atime` and `-ctime`. The flag `-atime` traces when the files were last accessed (i.e. read without being modified, for instance using **cat** command), while the flag `-ctime` traces when the file's metadata (permissions, name, location, etc.) were last time changed.

Example 7: Find all obsolete files in the specified directory(-ies) which were not accessed for more than 1 year.

```
find pathToDirectory(-ies) -type f -atime +365
```

The three frequently used flags `-mtime`, `-ctime` and `-atime` have the lowest resolution of 1 day. To perform the search with even finer time resolution in minutes, we need to use the flags `-mmin`, `-cmin` and `-amin`.

By default, the command **find** searches through all subdirectories of specified directory. If we start the search in some top level directory in the file hierarchy, the search can take forever. If we are sure that the targeted files are not deeper in the directory structure than a certain level, we can use flags `-maxdepth` and `-mindepth` to greatly optimize the search.

Example 8: Find all files with an extension '.pdf' in the specified directory, not going deeper than 2 levels in the subdirectory structure.

```
find pathToDirectory(-ies) -maxdepth 2 -type f -name "*.pdf"
```

Example 9: Find all files with an extension '.pdf' in the specified directory, by looking only in the subdirectories (i.e. not in the current directory, and not in subsubdirectories, subsubsubdirectories, etc.). The solution is:

```
find pathToDirectory(-ies) -mindepth 2 -maxdepth 2 -type f -name "*.pdf"
```

Finally, and very importantly, we describe the flag `-exec` which is used to specify the 'Action' part in the previously mentioned generic syntax, i.e. the command input which **find** needs to execute on the spot on the search outcome. The syntax for flag `-exec` is a bit peculiar, but there are essentially two important things to remember:

- `\;` : the command input after the flag `-exec` is determined this way
- `{}` : when used in combination with `-exec`, this is a placeholder for the found file or directory

Example 10: Find all empty files in the specified directories and for each of them prints its size.

```
find pathToDirectory(-ies) -type f -exec stat -c %s {} \;
```

From this example it is self-evident when and how we use the placeholder `{}` for the found file or directory in combination with `-exec` flag.

Example 11: Find all empty files in the specified directory, and delete them immediately:

```
find pathToDirectory(-ies) -type f -size 0 -exec rm {} \;
```

It is also possible to execute multiple commands on the files or directories which **find** has found, we just need to use separate `-exec` flag for each command input:

Example 12: Find all files in the specified directory, and for each of them: a) print the full metadata with **ls -al**; and b) print the size with **stat -c %s**.

```
find pathToDirectory(-ies) -type f -exec ls -al {} \; -exec stat -c %s {} \;
```

Equivalently we can use **while+read** construct in combination with the process substitution operator `<( ... )` to achieve the same result:

```
while read File; do
 ls -al $File
 stat -c %s $File
done < <(find pathToDirectory(-ies) -type f)
```

The second solution is more readable, less error prone and easier to generalize by adding more actions to the found files.

3. Online monitoring: tail -f

In general, we can view the whole content of the file with the **cat** command, or if we want paging to appear one screen at a time we can use commands like **more** or **less**. On the other hand, we can select and view only the select part of the file with commands like **sed**. For instance, if the starting file named 'example.txt' has the following content:

```
line 1
line 2
line 3
line 4
line 5
line 6
line 7
```

we have already seen in previous sections that we can select with **sed** for viewing only the lines in the specified range (both ends included), like in the following example:

```
sed -n 2,4p example.txt
```

Flag `-n` ensures that the starting file is not superimposed with the desired selected printout, while `p` stands for 'print'. The result is:

```
line 2
line 3
line 4
```

Alternatively, if we are interested to print only the first 'n' lines of the file, we can use **head -n** command. For instance:

```
head -3 example.txt
```

will print only the first 3 lines:

```
line 1
line 2
line 3
```

On the other hand, if we are interested to print only last 'n' lines of the file, we can use **tail -n** command. For instance, to get programmatically only the last line in the file, we can use:

```
tail -1 example.txt
```

which gives for the above example:

```
line 7
```

Without arguments, **head** and **tail** print by default the first and the last 10 lines, respectively.

Besides these simple use cases, the important non-trivial use case is provided with the flag **-f** of **tail** command. Namely, with **tail -f** we can monitor online the output of file as the file content gets updated. That means that if we have redirected the `stdout` stream of some command to a file, and if we execute **tail -f** on that file, we will monitor what that command is doing just as we are looking at its printout on the screen. However, what is non-trivial here is that we can execute **tail -f** on that file from any terminal and monitor online what that command is doing, not necessarily from the same terminal where the command is started. This is best illustrated with the following example:

```
( echo ${BASHPID}; while :; do echo "1: $(date)"; sleep 10s; done; ) 1>first.log
&
( echo ${BASHPID}; while :; do echo "2: $(date)"; sleep 20s; done; )
1>second.log &
( echo ${BASHPID}; while :; do echo "3: $(date)"; sleep 30s; done; ) 1>third.log
&
```

With this example, we have started three subshells in the background, where each of them after 10s, 20s and 30s, respectively, prints the time stamp, which is redirected via `1>` in its own log file. Since all 3 subshells are running in the background, we have the control over the terminal, and we can for instance checkout the status of submitted jobs:

```
jobs -l
[1]  20860 Running                 ( echo ${BASHPID}; while :; do echo "1:
$(date)"; sleep 10s; done ) > first.log &
[2]- 20867 Running                 ( echo ${BASHPID}; while :; do echo "2:
$(date)"; sleep 20s; done ) > second.log &
[3]+ 20873 Running                 ( echo ${BASHPID}; while :; do echo "3:
$(date)"; sleep 30s; done ) > third.log &
```

The great thing now is that we can see directly in the terminal what each of these jobs is doing in the background. For instance:

```
tail -f first.log
```

This gives:

```
20860
1: Tue Jun 23 12:39:57 CEST 2020
1: Tue Jun 23 12:40:07 CEST 2020
1: Tue Jun 23 12:40:17 CEST 2020
1: Tue Jun 23 12:40:27 CEST 2020
1: Tue Jun 23 12:40:37 CEST 2020
1: Tue Jun 23 12:40:47 CEST 2020
```

As the subshell execution proceeds, the output of **tail -f** gets updated on the screen automatically, just like the subshell is directly running in the terminal, and not in the background. If we now hit `Ctrl-C`, we terminate only the **tail -f** command, without any interference with the running subshell in the background. After terminating with `Ctrl-C`, we can inspect the status of second subshell running in the background by executing:

```
tail -f second.log
```

This gives:

```
20867
2: Tue Jun 23 12:40:02 CEST 2020
2: Tue Jun 23 12:40:22 CEST 2020
2: Tue Jun 23 12:40:42 CEST 2020
2: Tue Jun 23 12:41:02 CEST 2020
```

We now monitor online what the second subshell running in the background is doing. This functionality works from any terminal, since viewing the content of physical files with **tail -f** is not limited to any particular terminal.

4. Timing: timeout and time

Frequently in practice we are faced with the situation when the command execution gets stalled, without clear indication when its execution might resume. For instance, if we are copying files over the network, and if the network connection experiences a problem, the copying itself will hang until the network connection recovers. But for instance if are copying over network 1000 files containing our data, and if we managed to copy 90% of them, clearly we can reach the decent statistics and reliable results in our analysis, even if we did not analyse the whole dataset.

In general, we can prevent command to hang forever with the **timeout** command. This command in essence ensures that some command is run within a specified time limit. Its generic syntax is:

```
timeout someInterval someCommand
```

This syntax translates into the following use case: Start a command named 'someCommand', and terminate it if it still runs after the specified time interval 'someInterval'.

Again, we use command **sleep** to illustrate the use cases of **timeout** in concrete examples, but the examples below apply to any other command:

```
timeout 5s sleep 10s
```

This will terminate **sleep** command already after 5s, with non-zero exit status 124 (check the 'man' pages of **timeout**). The exit status is non-zero, since command failed to complete its execution within the specified time interval. By default, **timeout** terminates the command execution with `TERM` signal, but we can send any other supported signal (check out the list with **kill -l**) by using **-s** flag, for instance:

```
timeout -s KILL 5s sleep 10s
```

In the case command fails by itself within the specified time interval, then the exit status of **timeout** is the exit status of command.

Finally, in the case of successful command completion within the specified time interval, e.g.

```
timeout 20s sleep 10s
```

the exit status of **timeout** is 0.

As the last remark, we indicate that unfortunately command **timeout** can deal only with **Linux** commands, and not for instance with **Bash** functions.

In a completely different context, we use expression 'timing' when we want to summarize the usage of system resources by a given command. This is simply achieved with the command **time**. Its generic syntax for most cases of interest is very simple:

```
time someCommand
```

Here 'someCommand' is meant in a broader sense, and can be any **Linux** command, **Bash** built-in command, **Bash** function, etc. For instance:

```
time sleep 4s
```

gives the following expected printout:

```
real    0m4.002s
user    0m0.000s
sys 0m0.002s
```

However, we can use also **time** for **Bash** code snippets directly:

```
time for i in {1..1000}; do date; done > /dev/null
```

produces:

```
real    0m1.499s
user    0m0.165s
sys 0m1.397s
```

This is clearly a great utility when the efficiency of code execution starts to matter.

5. Counting: wc

The number of different elements (lines, words, characters) in the file content, or in the `stdout` of some command, can be conveniently obtained with the command **wc** ('word count').

For instance:

```
echo "a bb ccc" | wc
```

provides the following output:

```
      1       3       9
```

The first entry is the number of lines (1), then the number of words (3, namely "a", "bb" and "ccc"), and finally the number of characters (9, since both the empty characters and hidden new lines also count!). Using **wc** to count the number of characters frequently leads to surprises and is not recommended because the hidden new line character '\n' at the end of each line is also counted, for instance:

```
echo | wc
```

prints

```
       1       0       1
```

In the above printout, 1 character corresponds to the default new line character '\n' in **echo**.

For the counting of the number of lines and words this command behaves as expected. Typically, we just want number of lines or number of words, when flags **-l** or **-w** can be used.

The command **wc** can also count the elements of the physical file. For instance, if the starting file 'wcExample.txt' has the following content

```
line 1 a
line 2 bb b
line 3 c ccc cc
```

then we can use the following syntax:

```
wc -l < wcExample.txt
# prints 3, total number of lines
wc -l < wcExample.txt
# prints 12, total number of words
```

6. Building programmatically command input: eval

We have already seen how we can programmatically provide the arguments to the commands by referencing the content of some variables with the general syntax `${var}`. Now we generalize this idea and illustrate how we can build the whole command input programmatically, including even the pipes. This can be achieved by using the **Bash** built-in command **eval**. Some experts warn against its usage due to potential security holes and argue that this command shall be instead renamed into 'evil'. Typically, the command **eval** can be used to force additional re-evaluation of command input, if its interpretation ended up in some intermediate state.

In essence, the command **eval** enforces the command-line processing once again. It's a very powerful feature, which enables to write scripts that create command string on-the-fly and then pass it to **Bash** for execution. By using this mechanism, the **Bash** scripts can for instance modify their behaviour when they are already running.

We start with the following example, where from the **date** command we extract only the hour, minutes and seconds:

```
date | awk '{print $4}'
# prints 12:22:02
```

But now let us attempt in the **Bash** variable **DateSimple** to store that command input:

```
DateSimple="date | awk '{print $4}'"
```

If we now attempt to reference the content of the variable **DateSimple**, and use it directly in the same way as command input, we get an error:

```
$DateSimple
# date: extra operand ‘awk’
# Try 'date --help' for more information.
```

However, this saves the day:

```
eval $DateSimple
```

What happened above is the following: Upon expanding the content of variable **DateSimple**, **Bash** interpreted pipe `|` and `awk` as arguments to **date** command, and since the command **date** can handle only one argument (besides flags which are indicated with prepended `-` or `--`), it bailed out when it hit at the second argument, which is string `awk`. When interpreting the command input, one of the very first thing **Bash** is looking for are pipes `|`, however, since in the literal command input **$DateSimple** there are no pipes (before expansion!), **Bash** stopped searching for them. Then after expanding **$DateSimple**, **Bash** continued with the other steps in the command input interpretation, none of which includes pipes. Therefore, the pipe `|` ended up being interpreted as a mere argument to **date** command. This sort of problems can be fixed with **eval**, because that commands literally forces the command input re-interpretation from scratch. After using **eval $DateSimple**, and after expanding **$DateSimple**, **Bash** goes from scratch through the command input interpretation, and interprets the pipe `|` correctly.

To a certain degree, echoing the command input into the file, and then sourcing that file, achieves the same functionality as **eval**, but it is much less efficient.

Finally, we mention the following frequently encountered example. We can generate sequences with brace expansion, for instance:

```
echo {0..9}
```

prints

```
0 1 2 3 4 5 6 7 8 9
```

But what if we want to pass low and upper ends via variables? We could try:

```
Min=0
Max=9
echo {$Min..$Max}
```

The result is however only the following printout:

```
{0..9}
```

This is a nice example where **eval** saves the day again, because

```
eval echo {$Min..$Max}
```

forces the re-interpretation of intermediate result `{0..9}`, and produces the desired output:

```
0 1 2 3 4 5 6 7 8 9
```