Some linux tricks

Yves-Henri Sanejouand

November 15, 2018

Bash line commands

awk

To manipulate a file with well defined fields (*e.g.* columns).

Examples:

```
Prompt> awk '{ if ( substr($1,1,5) == "Keywd" ) \
        print substr($2,2,length($2)-1) ; else print $3 }' file
```

Prints end of second field of each line of the *file* if first field starts with *Keywd*; prints third field otherwise. *1,5* means: substring starts at first character and is five characters long.

```
Prompt> awk '{ if ( $1 > 0 ) { n=n+1 ; print n } \
        else print $1 }' file
```

Prints the current number of lines found with a positive value of the first field. Otherwise, prints its value.

```
Prompt> awk -v x="$value1" -v y="$value2" '{ print x*y*$1 }' file
```

Prints first field of *file* multiplied by the product of *$value1* and *$value2*. Quoting variables is recommended (in case of leading blanks).

```
Prompt> awk -F'X' '{ print NF-1 }' file
```

Counts and prints the number of times word X is found in each line of the *file* (*-F* provides the separator between fields and *NF* is an **awk** variable giving the number of fields in the current line).

```
Prompt> awk -F'[_=\ ]' '{ print $NF }' file
```

Prints the last word of each line of the *file*, words being separated either by an underscore, the sign equal or a single blank.

```
Prompt> awk '{ if ( $1 == "Keywd" ) n=n+1 } END { print n }' file
```

Prints the number of times *Keywd* is found in first column of *file*. *END* means: wait for the end of *file* before doing the next action.

```
Prompt> awk '{ $2=$2+1 ; print FILENAME ": " $0 }' file
```

Adds one to each element of the second field of the *file* and prints the filename before each modified line (*$0*). *FILENAME* is an **awk** variable.

```
awk '{ if ( substr($1,1,1) == "!" ) \
     { f=FILENAME ; n=split(f,wf,"/") ; print wf[n], $2 }}' file
```

Prints filename without the directory (like **basename**), followed by second field for all lines starting with *!*.

```
awk '{ if ( substr($1,1,1) == "!" ) { gsub("x","y",$1) ; print } \
       else print }' file
```

Substitutes x by y in first field of all lines starting with *!*.

```
awk '{ if ( $0 ~ "<<" ) f=0 }{ if ( $0 ~ ">>" ) f=1 } \
     { if ( f == 1 ) print }' file
```

Prints all lines after those containing `>>` and before those containing `<<`.

```
Prompt> awk '{ a[$1]=a[$1]+1 } END \
      { for( i in a ) print i, a[i] }' file
```

Prints the number of times the different first fields are found in *file*.

```
Prompt> awk '{ for ( i=1 ; i <= NF ; i++ ) { print $i }}' file
```

Prints all fields of *file* in a single column.

```
Prompt> awk '{ OFS="," ; print $1, $2, $3 }' file
```

Prints three first fields, separated by commas. *OFS* is the **awk** variable that provides the separator on output.

```
Prompt> awk '{ ORS=" " ; print $1 }' file
```

Writes the first column of *file* in a single line. *ORS* is the **awk** variable that provides the end-of-line ($\backslash n$ by default).

```
Prompt> awk '{ printf "%-10s%4s%7.2f\n", $1, $2, $3 }' file
```

Formatted output for two chains of characters followed by a real number. By default: righ-justified, but it can also be left justified (as in first field). $\backslash n$ (new line) needs to specified.

```
Prompt> awk '{ print > "file"$1 }' file
```

Prints lines in different files, their suffix being the first field.

```
Prompt> awk 'BEGIN { print ( 0.1 > 1e-5 ) }'
```

Prints 1, since the test is true. Would print 0 otherwise. Note that **awk** handles the scientific notation, at variance with **bc**.

Nota bene: Conditional statements are loosely checked by **awk** and error messages are scarce. As a result, syntax errors can have major consequences. *Tip:* The field separator can be a whole word or a regular expression.

basename

To get the main part of a filename, without the directory.

Example:

```
Prompt> basename file suffix
```

Also removes the suffix, that is, the end of the filename, if it ends by *suffix*.

Nota bene: useful in scripts.

date

To change the date of the system.

Example:

```
Prompt> date -s '2017-06-27 11:31:00'
```

```
Prompt> date -s '11:31:00'
```

Changes only the hour.

Nota bene: For root users.

find

To find file(s).
Major tip: a command can be executed for each of them.

Examples:

```
Prompt> find your-directory -name \*.html \
       -exec grep -H 'searched_key' {} \;
```

Starting from your-directory, finds files recursively, looking for (**grep**) *searched_key* inside files whose name ends with *.html* (*-H* adds the filename to the output).

```
Prompt> find . -name \*.html -exec cp {} . \;
```

Starting from current directory (.), copies (**cp**) recursively the files whose name ends with *.html* to current directory.

```
Prompt> find . -name \*.html -exec sed -i 's/searched_key/new_key/g' {} \;
```

Modifies files recursively, replacing (**sed**) *searched_key* by *new_key* inside files whose name ends with *.html*.

```
Prompt> find . -name \*.html -exec sh -c "grep 'searched_key' {} | \
        grep 'other_key'" \;
```

Prints lines with both *searched_key* and *other_key*. Going through **sh** allows to execute a series of commands for each file whose name ends with *.html*.

Nota bene:
{} means: for each file found.
The backslashes before "*" and ";" are required, in order to avoid their prior interpretation by the shell. *Tip:* Quotes also work.
Tip: Just to find files, consider **locate**, which is quicker.

grep

To find strings inside files.

Examples:

```
Prompt> grep -f file1 file2
```

For each string found in each line of *file1*, prints the lines of *file2* with the string.

```
Prompt> grep -o -f file1 file2 | grep -v -f - file1
```

Prints the strings of *file1* not found in *file2* (*-o*: prints only the string found in *file2*).

```
Prompt> grep -o -b string file
```

For each *string* found in the *file*, prints a line with the *string* and the byte where it starts (*-b*).

```
Prompt> grep -n string file
```

Prints the line number followed by the line with the *string*.

```
Prompt> grep -n -- negative-number file
```

Prints the line numbers of *file* with the *negative number* (- - means that what follows is not interpreted as an option).

```
Prompt> grep -A5 --no-group-separator string file
```

Prints the lines of *file* with the *string*, and the five (*-A5*) following ones, with no separator after each series of six (1+5) lines.

join

To fuse a pair of files that have common keys.

Examples:

```
Prompt> join file1 file2
```

This prints the content of *file1* and *file2* when the same key is found in their first columns.

```
Prompt> join -1 2 -2 3 file1 file2
```

Here, keys are expected in column two of *file1* and column three of *file2*.

```
Prompt> join -t',' -a 1 -a 2 -e NULL -o 0,1.2,2.2 file1 file2
```

To print *NULL* when the key is missing in *file1* (*-a* 2) or *file2* (*-a* 1). The *-e* option works only when the *-o* (output format: file number before the dot, column number after) one is specified (*-t* provides the separator on output).

```
Prompt> join -t',' -a 1 -a 2 -e NULL -o auto file1 file2
```

Keeps all fields on output (*auto* is a rather recent option).

Nota bene: Keys have to be in same order (e.g. **sort** -k) in both files.

paste

To join files side by side, or to join the lines of a given file.

Examples:

```
Prompt> paste -d';' file*
```

Prints files side by side (*-d* is the separator).

```
Prompt> paste -s -d'  \n' file
```

Adds a blank at the end of the first two lines instead of a carriage return.

ps

To get information about running processes.

Example:

```
Prompt> ps -eo pid,user,lstart,cmd
```

Show the process identifier (*pid*), the starting time (*lstart*)...

rename

To rename file(s).

Example:

```
Prompt> rename 's/pdb/ent/' *.pdb
```

Changes the name of all files whose name ends with *.pdb*.

Nota bene:
The syntax for the substitution like with **sed** or **vi**.
This means that, in the above example, only the first "pdb" string found is changed. For instance, a file named pdbxxx.pdb is renamed as follows: entxxx.pdb.

SECONDS

A built-in bash variable counting seconds.

Example:

```
Prompt> SECONDS=0 ; bash-command ; echo $SECONDS
```

Prints how long the *bash-command* lasted.

sed

To edit a file from outside.

Examples:

```
Prompt> sed -i 's/chain/other/g' file
```

Substitutes all instances of *chain* by *other* in *file*. The syntax is like in **vi**. *-i* means: in file (–in-place).

```
Prompt> sed -i '/string /d' file
```

Deletes (*d*) all lines with *string* in *file*.

```
Prompt> sed -i.bak '9,12d' file
```

Deletes lines *9* to *12*, the original *file* being saved with a *.bak* suffix.

```
Prompt> sed -n '/string1/,/string2/p' file
```

Prints lines between *string1* and *string2*.

```
Prompt> sed -n -e '1,12p' -e '15p' file
```

Prints (*p*) lines *1* to *12* and *15*. *-e*: allows to specify multiple commands.

sort

To sort a file by columns.

Example:

```
Prompt> sort -k1,1 -k3,3n file
```

file is sorted using column 1 and then column 3, by integers (*n*) in the latter case.

wget

Gets files or directories through the internet.

Examples:

```
Prompt> wget http\://www.remote-host.com/remote-file
```

Gets *remote-file*.

```
Prompt> wget -r http\://www.remote-host.com
```

Gets the whole site (*-r*: recursively).

```
Prompt> wget -r -l2 http\://www.remote-host.com/remote-directory
```

Gets *remote-directory* and its sub-directories (*-l2*: two sub-levels).

xargs

Provides arguments one by one.
Tip: For commands that do not accept more than one argument.

Examples:

```
Prompt> ls *.tar | xargs -I {} tar xf {}
```

{} is replaced by each argument provided by **xargs**, one after the other, and all files found in the local tar-files (**.tar*) are extracted.

```
Prompt> cat file | xargs -I {} cp ../file-directory/{} \
        other-directory
```

Copies files that are in *file-directory* and whose names are listed in *file*.

```
Prompt> ls *.pdb | xargs -n 1 cp -t other-directory
```

Or listed (with another syntax). In both cases, **xargs** provides one argument per line to **cp**, which copies them to *other-directory*.

```
Prompt> cat file | xargs -I {} sh -c 'mv {}* other-directory'
```

For expanding the wildcard, it is necessary to go through **sh** (here, only the beginning of the filenames is found in *file*).

Latex tools

makeindex

Makes an index for a Latex file.

In the preamble of the Latex file:

```
\usepackage{makeidx}
```

```
\makeindex
```

In the text:

```
\index{Indexed-word}
```

To gather them by topic:

```
\index{Indexed-topic!Indexed-word}
```

Then, where you want to see the index printed:

```
\printindex
```

To actually see it, you need to run **Latex**, then **makeindex**.
This can be done through **texmaker**.

texcount

Counts words in a Latex file. That is: Latex commands are ignored.

Example:

```
Prompt> texcount -sum myfile.tex
```

FILE: myfile.tex
Sum count: 2373
Words in text: 2261
Words in headers: 97
Words in float captions: 4
Number of headers: 25
Number of floats: 0
Number of math inlines: 11
Number of math displayed: 0

```
Prompt> texcount -v -sum myfile.tex
```

Checks what it is doing.

Index