

Some linux tricks

Yves-Henri Sanejouand

December 4, 2023

Bash line commands

awk

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

Examples:

```
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 of the *file*. Otherwise, prints its value.

```
Prompt> awk 'BEGIN { s=0 } { if ( NF > 0 ) s=s+$1+1 } \
           END { print s }' file
```

Prints the sum of first fields (+1). After *BEGIN*: the command executed before reading the file. After *END*: the command executed after reading the file. *NF* is an **awk** variable giving the number of fields in the current line.

```
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 -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.

```
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 -F'_|=| AND ]' '{ print substr($NF,2) }' file
```

Prints the last word of each line of the *file*, except the first character, words being separated either by an underscore, the sign equal or the word AND (with blanks on both sides).

```
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. Note that **grep** would print the matches whatever the column.

```
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> echo $word | awk -F'.' '{ OFS="." ; $NF="txt" ; print $0 }'
```

Replaces the last string after a dot by *txt* in variable *word*.

```
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: right-justified, but it can also be left justified (as in first field). $\backslash n$ (new line) needs to be 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**. However, beware of overflows.

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. Note that when it is *not* a blank, blanks are included in words.

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 -type f -mtime -60 -mtime +1
```

Starting from *your_directory*, looks recursively for files (*-type f*) modified between yesterday (*+1*) and sixty days ago.

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

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" ! -path './not_there/*' \  
-exec cp -u {} your_directory/ \;
```

Copies found files in *your_directory*, except those from directory *not_there*, and only if they are newer (*-u*) than those already there.

```
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 -o . file
```

Prints all characters, with a single one per line.

```
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.

```
Prompt> grep --color='auto' -P -n "[\x80-\xFF]" file
```

Prints the lines of *file* where there are non-ASCII characters and highlights them (*-P* means that perl regular expressions are used).

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
```

Joins three consecutive lines.

pgrep

To get information about running processes.

Example:

```
Prompt> pgrep -P $$
```

Shows subprocess identifiers.

Nota bene: useful in scripts.

ps

To get information about running processes.

Example:

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

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

RANDOM

To get a random integer.

Example:

```
Prompt> echo $(( 1 + RANDOM % 10 ))
```

Provides an integer between 1 and 10.

Nota bene: There is also a built-in bash variable (`$RANDOM`).

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 '9,12d;13q' file
```

Deletes lines *9* to *12*, then quit on next line. Can save time in the case of large files.

```
Prompt> sed '1i Your text' file
```

Inserts a first line with *Your text*.

```
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.

Examples:

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

file is sorted using column 1 and then column 3, by integers (*n*) in the latter case. Last five lines are shown (**tail**).

```
Prompt> sort -V -k1 file
```

Column 1 of *file* is sorted according to numbers within the text.

tee

Appends input to output(s).

Example:

```
Prompt> echo yhs | tee -a *.dat
```

Adds a line with *yhs* to all files with a *dat* suffix.

wget

Gets files or directories through the internet.

Examples:

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

Gets *remote_file*.

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

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

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

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

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

ASCII, 6
awk, 1
 FILENAME, 2
 NF, 2
 OFS, 3
 ORS, 3

basename, 3
bash variables
 \$RANDOM, 8
 \$SECONDS, 8
bc, 3

cat, 10
cp, 5, 10

date, 4

find, 5

grep, 5, 6

Internet, 10

join, 7

Latex, 11
 Count, 11
 Index, 11
 makeindex, 11
 Preamble, 11
 Words, 11
locate, 5
ls, 10

paste, 7
perl, 6
Process, 7
ps, 7

RANDOM, 8

rename, 8
Root, 4

sed, 5, 8, 9
Separator, 3, 6, 7
sh, 5, 10
sort, 7, 9

tail, 9
tar, 10
texcount, 11
texmaker, 11

vi, 8, 9

wget, 10

xargs, 10