

Abims

Advanced Linux ABiMS

Training Module 2017

Mark Hoebeke
Philippe Bordron
Gildas Le Corguillé

UPMC
SORBONNE UNIVERSITÉS



So now you're **the** Linux gal/guy of the lab. Congrats, colleagues keep coming to you for advice (at best) or for a helping hand (i.e. burden you with chores they can't take care of themselves).

How do I:

- ~~Get rid of them?~~ (not an option)
- Extract relevant information from the **myriad humongous** files ?
- Run a **series of commands** and make data **flow** between them ?
- Write **command files** containing lists of commands operating on data files ?

- 1 A Quick Refresher**
- 2 Redirections & Pipes**
- 3 Slicing 'n Dicing Files**
- 4 Regular Expressions**
- 5 Awk 101**
- 6 Batch Files 101**

- 1 A Quick Refresher**
- 2 Redirections & Pipes
- 3 Slicing 'n Dicing files
- 4 Regular Expressions
- 5 Awk 101
- 6 Batch Files 101

Where am I ?

```
[stage01@nz~]$ pwd
```

Which files/directories are located "here" ?

```
[stage01@nz~]$ ls  
[stage01@nz~]$ ls .
```

Which files/dirs are located in /tmp (with full details and hidden files) ?

```
[stage01@nz~]$ ls -la /tmp
```

How do I get to /tmp (make /tmp my current directory) ?

```
[stage01@nz~]$ cd /tmp
```

How do I create directories `~/foo/bar/baz` ?

```
[stage01@nz~]$ mkdir -p ~/foo/bar/baz
```

How do I copy file `quux` to `~/foo/bar/baz` ?

```
[stage01@nz~]$ cp quux ~/foo/bar/baz
```

How do I *move* file `corge` to `~/foo/bar/baz` ?

```
[stage01@nz~]$ mv corge ~/foo/bar/baz
```

How do I copy *directory* `grault` to `~/foo/bar/baz` ?

```
[stage01@nz~]$ cp -r grault ~/foo/bar/baz
```

How do I remove (delete forever) file garply ?

```
[stage01@nz~]$ rm garply
```

How do I remove directory waldo (with all its contents) ?

```
[stage01@nz~]$ rm -rf waldo
```

How do I remove *empty* directory fred ?

```
[stage01@nz~]$ rmdir fred
```

Quick Refresher | CLI Survival Kit

How do I know what kind of data is stored in file `plugh` ?

```
[stage01@nz~]$ file plugh
```

How do I display the contents of file `xyzzzy` (and recover control of the terminal rightaway) ?

```
[stage01@nz~]$ cat xyzzzy
```

How do I display the beginning (end) of file `thud` ?

```
[stage01@nz~]$ head thud  
[stage01@nz~]$ tail thud
```

How do I page through file `ioofa` ?

```
[stage01@nz~]$ less ioofa
```

How do I edit file `omtg` ?

```
[stage01@nz~]$ gedit omtg
```


How do I run program `jimbo` in the background ?

```
[stage01@nz~]$ jimbo &
```

How do I relegate already running program `wharty` to the background?

```
[stage01@nz~]$ wharty  
[Ctrl+Z]  
(...)  
[stage01@nz~]$ bg
```

Quick Refresher | The Ground Rule

When in doubt about running a program :

**READ
THE
FINE
MANUAL**

```
[stage01@nz~]$ man command
```

1. Open a terminal and connect to `nz`

```
[stage01@nz~]$ ssh -Y nz
```

2. Jump to one of the cluster nodes **(nobody runs jobs on nz !)**

```
[stage01@nz~]$ qlogin
```

3. Go to your "project" directory **(don't work in you home directory !)**

```
[stage01@nz~]$ cdprojet
```

4. Get the course material

```
[stage01@nz~]$ wget https://frama.link/aldataset
```

5. Unpack the course material

```
[stage01@nz~]$ tar -jxvf Linux-Avance.tbz
```

- 1 A Quick Refresher
- 2 Redirections & Pipes**
- 3 Slicing 'n Dicing files
- 4 Regular Expressions
- 5 Awk 101
- 6 Batch Files 101

Displaying command output on the terminal has its limitations :

- 1. Scrolling capacity is finite**
- 2. Difficult to reuse for further processing**

The puzzle :

- How do I build the list of files in the current directory matching a specific pattern and modified at a given date ?

Some of the pieces :

- I know how to list the files in the current directory (with `ls`)**
- I know how to look for patterns in text files (with `grep`)**

What's missing :

- I don't know how to feed the output of `ls` into `grep`

Redirections & Pipes | Redirections

All programs generate their output into *channels* (special types of files). The terminal is just the default output channel (**stdout** for standard output).

Linux gives us **redirections** to replace the default output channel with a file.

Ex.: Using a redirection to store the output of a command to a file

```
[stage11@nz ~]$ ls -l * > listoffiles.txt
[stage11@nz ~]$ cat listoffiles.txt
-rwxr-xr-x 1 mhoebeke mhoebeke      55 sept.  5 2012 acteur.tab
-rwxr-xr-x 1 mhoebeke mhoebeke    488 sept.  5 2012 address.tab
-rwxr-xr-x 1 mhoebeke mhoebeke  30768 sept.  5 2012 annuaire.csv
(...)
```

The redirection character **>** added after a command and its arguments and **followed by a filename** will create a file containing the output of the command.



If the file already exists, it will be overwritten

Redirections & Pipes | Redirections

Programs can read their input from *channels* (special types of files). There is a default input channel (**stdin** for standard input). Linux gives us **redirections** to use a file as standard input.

Ex.: Using a redirection to use a file as input for grep using a redirection

```
[stage11@nz ~]$ grep tab < listoffiles.txt  
-rwxr-xr-x 1 mhoebeke mhoebeke      55 sept.  5 2012 acteur.tab  
-rwxr-xr-x 1 mhoebeke mhoebeke     488 sept.  5 2012 address.tab  
-rwxr-xr-x 1 mhoebeke mhoebeke 1315419 sept.  5 2012 insulin.vs.nt.blastn.tab  
(...)
```

The redirection character **<** added after the arguments of a command and **followed by a filename** will use the file as input for reading data instead of **stdin**.

Redirections & Pipes | Redirections

Input and output redirections can be combined.

Ex.: Using a redirection to use a file as input for `grep`, and for storing the result in a file

```
[stage11@nz ~]$ grep tab < listoffiles.txt > tabfiles.txt  
[stage11@nz ~]$ cat tabfiles.txt  
-rwxr-xr-x 1 mhoebeke mhoebeke      55 sept.  5 2012 acteur.tab  
-rwxr-xr-x 1 mhoebeke mhoebeke    488 sept.  5 2012 address.tab  
-rwxr-xr-x 1 mhoebeke mhoebeke 1315419 sept.  5 2012 insulin.vs.nt.blastn.tab  
(...)
```


Redirections & Pipes | Pipes

Pipes can be used to directly channel **stdout** from one command into **stdin** of the next command

Ex.: Using a pipe to **grep** for a pattern in the output of **ls**

```
[stage11@nz ~]$ ls -l | grep tab
-rwxr-xr-x 1 mhoebeke mhoebeke      55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 mhoebeke mhoebeke     488 sept.  5  2012 address.tab
-rwxr-xr-x 1 mhoebeke mhoebeke  1315419 sept.  5  2012 insulin.vs.nt.blastn.tab
(...)
```

The pipe symbol **|** is placed after the arguments of the first command and before the second command.

Series of commands can be linked with pipes.

```
[stage11@nz ~]$ ls -l | grep tab | grep -v insulin
-rwxr-xr-x 1 mhoebeke mhoebeke      55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 mhoebeke mhoebeke     488 sept.  5  2012 address.tab
(...)
```

There is a special channel, **stderr** (for standard error), **different from stdout**, where commands write error messages when necessary.
By default stderr is also the terminal output...

Ex.: Redirecting `stdout` only will still generate error messages on the terminal.

```
[stage11@nz ~]$ ls -lR /home/fr2424 > /tmp/lsfr2424.txt
ls: cannot open directory /home/fr2424/administration/bmasse: Permission denied
ls: cannot open directory /home/fr2424/administration/carou: Permission denied
ls: cannot open directory /home/fr2424/administration/hmignot: Permission denied
(...)
```

The redirection of **stderr** is possible by adding the **>&** redirection symbol after a command's arguments.

```
[stage11@nz ~]$ ls -lR /home/fr2424 > /tmp/lsfr2424.txt && /tmp/lseerrors.txt
[stage11@nz ~]$ cat /tmp/lseerrors.txt
ls: cannot open directory /home/fr2424/administration/bmasse: Permission denied
ls: cannot open directory /home/fr2424/administration/carou: Permission denied
ls: cannot open directory /home/fr2424/administration/hmignot: Permission denied
(...)
```

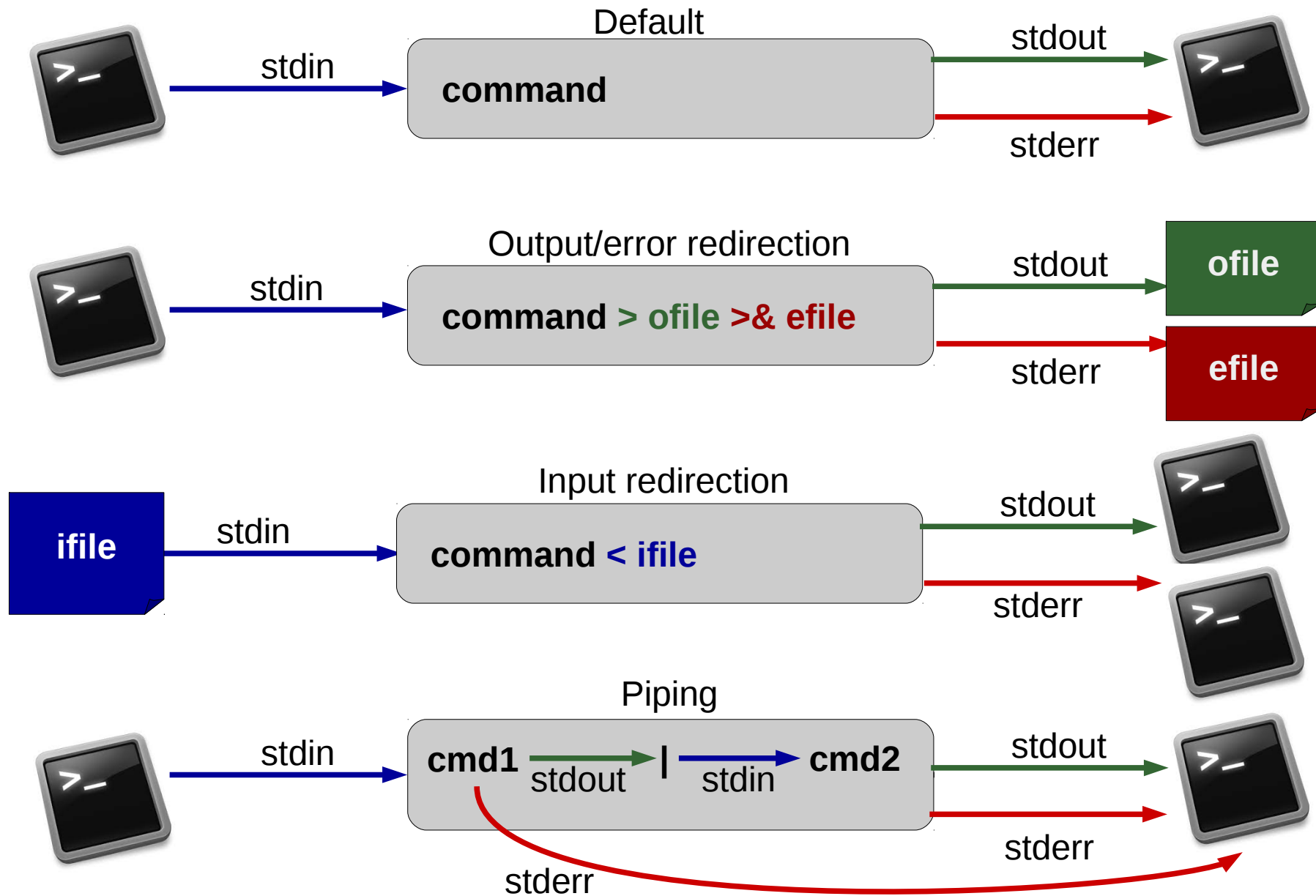
To ignore what's generated on an output channel (**stdout** or **stderr**), it can be redirected to a special file : **/dev/null**.

```
[stage11@nz ~]$ ls -lR /home/fr2424 > /tmp/lsfr2424.txt >& /dev/null  
[stage11@nz ~]$
```

To redirect an output channel (**stdout** or **stderr**) to an already existing file without overwriting its contents, the redirect append (**>>**) symbol can be used.

```
[stage11@nz ~]$ ls *.tab > fileswithcolumns.txt  
[stage11@nz ~]$ wc -l fileswithcolumns.txt  
5 fileswithcolumns.txt  
[stage11@nz ~]$ ls *.csv >> fileswithcolumns.txt  
[stage11@nz ~]$ wc -l fileswithcolumns.txt  
7 fileswithcolumns.txt
```

Redirections & Pipes | Summary



- 1 A Quick Refresher
- 2 Redirections & Pipes
- 3 Slicing 'n Dicing files**
- 4 Regular Expressions
- 5 Awk 101
- 6 Batch files 101

Slicing 'n Dicing | Foreword

**Beware when copying text files from foreign systems, especially from the MS-DOS family tree (including the Windows offspring).
Format differences can bite real hard.**

A typical symptom of format discrepancy is :

```
[stage01@nz~]$ ./phyml-mpi-multi.sh  
-bash: ./phyml-mpi-multi.sh: /bin/sh^M bad interpreter:  
No such file or directory
```

Doh ! A DOS line terminator !

The cause of the disease :

- Linux/Unix uses a single character to signal "newline" \n (line feed).
- DOS-ish systems use two : \r (carriage return) and \n.



And the cure relies on the dos2unix command :

```
[stage01@nz~]$ dos2unix ./phyml-mpi-multi.sh
```

Slicing 'n Dicing | Extracting Lines

Remember grep ?

The `grep` command takes two arguments : a *pattern* and a *file name* ; it displays every line of the file matching the pattern.

```
[stage11@nz ~]$ grep ">" insulin.fas
>gi|163659904|ref|NM_000618.3| Homo sapiens insulin-like growth factor 1 (somatomedin C) (IGF1),
transcript variant 4, mRNA
(...)
```

`grep` has loads of options, among which the most common are :

- i (ignore upper/lower case differences),
- v (display lines *not* matching the pattern),
- c (display the line count instead of the actual lines),
- r (recursively examine the contents of the **directory** given as second argument).

```
[stage11@nz ~]$ grep -r -c -i TRANSCRIPT .
./insulin_vs_nt.blast:144
./acteur.csv:0
./insulin.fas:5
```

Contextual grep

When the relevant information spans several lines, **grep** can give contextual information. With the **-A *n*** option, **grep** displays for each matching line, the ***n* following** lines (A : after)

```
[stage11@nz ~]$ grep -A 1 ">" insulin.fas
>gi|163659904|ref|NM_000618.3| Homo sapiens insulin-like growth factor 1 (somatomedin C) (IGF1),
transcript variant 4, mRNA
TTTTGTAGATAAATGTGAGGATTTTCTCTAAATCCCTCTTCTGTTTGCTAAATCTCACTGTCCTGCTAA
--
>gi|163659900|ref|NM_001111284.1| Homo sapiens insulin-like growth factor 1 (somatomedin C) (IGF1),
transcript variant 2, mRNA
GCATACCTGCCTGGGTGTCCAAATGTAAGTAGATGCTTTCACAAACCCACCCACAAAGCAGCACATGTT
--
(...)
```

With the **-B *n*** option, **grep** displays for each matching line, the ***n* preceding** lines (B: before)

With the **-C *n*** option, **grep** displays for each matching line, the ***n* surrounding** lines (C: context)

```
[stage11@nz ~]$ grep -C 3 TIGR01365 iprscan.xml
<category>Biological Process</category>
<description>biosynthetic process</description>
</classification>
<match id="TIGR01365" name="serC_2: phosphoserine aminotransferase" dbname="TIGRFAMs">
  <location start="29" end="400" score="2.1e-214" status="T" evidence="HMMTigr" />
</match>
</interpro>
(...)
```


Slicing 'n Dicing | Extracting Columns

The `cut` command takes an option describing how to extract columns (aka fields) and an argument with the name of the file containing *tabular* (columns are separated by `<TAB>` characters).

Ex. : extracting the first column of a file using the `-f 1` syntax.

```
[stage11@nz ~]$ cut -f 1 acteur.tab
Chuck
Sylvester
Steven
(...)
```

Ex. : extracting the second and third columns of a file using the `-f 2,3` syntax.

```
[stage11@nz ~]$ cut -f "2,3" acteur.tab
Norris      72
Stallone    66
Seagal      61
(...)
```

Ex. : extracting **all but** the second and third columns of a file using the `--complement` option.

```
[stage11@nz ~]$ cut --complement -f "2,3" acteur.tab
Chuck
Sylvester
Steven
(...)
```

Ex. : using the `-d` syntax to specify the field delimiter.

```
[stage11@nz ~]$ cut -d ";" -f 2 annuaire.csv
Clio
Brice
Mathilde
(...)
```

The `sort` command is used to sort files. It takes a filename as argument and options allow to specify sort fields and/or sort types.

Ex. : alphabetically sorting the lines of a file

```
[stage11@nz ~]$ sort pop_ville.tab  
Paris      4193031  
Roscoff    3705  
Tokyo      13010279
```

Ex. : alphabetically sorting the lines of a file using a specific field (`-k` option)

```
[stage11@nz ~]$ sort -k 2 pop_ville.tab  
Tokyo      13010279  
Roscoff    3705  
Paris      4193031
```

Ex. : numerically sorting the lines of a file using a specific field (`-n` option)

```
[stage11@nz ~]$ sort -n -k 2 pop_ville.tab  
Roscoff    3705  
Paris      4193031  
Tokyo      13010279
```

Ex. : reversing the sort order with the `-r` option

```
[stage11@nz ~]$ sort -r -n -k 2 pop_ville.tab  
Tokyo      13010279  
Paris      4193031  
Roscoff    3705
```

The `uniq` command is used to remove consecutive identical lines in a file. On a sorted file, it removes all repeated lines.

```
[stage11@nz ~]$ wc -l condition1_sorted.go
44 condition1_sorted.go
[stage11@nz ~]$ uniq condition1_sorted.go
GO:0000166      nucleotide binding
GO:0003824      catalytic activity
GO:0005488      binding
... [11 lines total]
```

`uniq` can also be used to count occurrences with the `-c` option:

```
[stage11@nz ~]$ uniq -c condition1_sorted.go
  2 GO:0000166 nucleotide binding
  1 GO:0003824 catalytic activity
  7 GO:0005488 binding
(...)
```

or to extract unique occurrences with the `-u` option:

```
[stage11@nz ~]$ uniq -u condition1_sorted.go
GO:0003824      catalytic activity
GO:0005623      cell
GO:0006810      transport
GO:0008152      metabolic process
```

The `join` command is used merge two files having a *sorted* column in common.

It is used as follows :

```
join -1 n -2 m file1 file2
```

where :

- in `-1 n` : `n` is the position of the common column in `file1`
- in `-2 m` : `m` is the position of the common column in `file2`

```
[stage11@nz ~]$ head -1 acteur_sorted.tab
Chuck Norris 72
[stage11@nz ~]$ head -1 address_sorted.tab
Canet Guillaume Artmedia 20, Avenue Rapp 75007 Paris France
[stage11@nz ~]$ join -i -1 2 -2 1 acteur_sorted.tab address_sorted.tab
Norris Chuck 72 Chuck Box 872 Navasota, TX 77868 USA
Stallone Sylvester 66 Sylvester Rogue Marble Productions, Inc. 21731 Ventura Blvd.
Suite 300 Woodland Hills, CA 91364 USA
```

The `-i` option can be added to ignore case differences in key column values

Slicing 'n Dicing | Simple Text Substitutions

The **sed** command is the swiss army-knife for performing manipulation on the contents of (text) files. Its basic usage looks like :

```
sed "operation" [file]
```

Where :

- ***operation*** : recipe(s) describing operations to perform on the contents (substitute, delete, paste...)
- **file** : the file to act upon (optional : remember how pipes work ?)

Ex. : Simple text substitution

```
[stage11@nz ~]$ sed "s/Roscoff/Rosko/" pop_ville.tab
```

```
Rosko 3705  
Paris 4193031  
Tokyo 13010279
```

s/Roscoff/Rosko/

s : the **substitute** operation

Roscoff : the text we want to replace

Rosko : the replacement text

Written like this :

- **case sensitive** (roscoff ≠ Roscoff)
- only the **first occurrence** of a line is replaced

Ex. : Field delimiter substitution

```
[stage11@nz ~]$ sed "s/\t/;/g" acteur.tab  
Chuck;Norris;72  
Sylvester;Stallone;66  
Steven;Seagal;61
```

s/\t/;/g

s : the **substitute** operation

\t : the text we want to replace = the TAB character

; : the replacement text

g : a **flag** to indicate global substitution (all occurrences of the line)

The **i flag** can be used to ignore uppercase/lowercase differences on the pattern to match

Ex. : Using locations to operate on specific line ranges

```
[stage11@nz ~]$ sed "2,3s/\t/;/g" acteur.tab  
Chuck Norris 72  
Sylvester;Stallone;66  
Steven;Seagal;61
```

`2,3s/\t/;/g`

`2,3` : only apply the (substitution) operation on lines 2 to 3

Having fun with `sed`

Ex. : Using the delete operator

```
[stage11@nz ~]$ sed "2d" acteur.tab  
Chuck Norris 72  
Steven Seagal 61
```

Ex. : Combining operators : pasting & replacing

```
[stage11@nz ~]$ sed "2p; s/Sylvester/Sly/" acteur.tab  
Chuck Norris 72  
Sylvester Stallone 66  
Sly Stallone 66  
Steven Seagal 61
```

- Extract the actors last names from `acteur.tab`

```
[stage11@nz ~]$ cut -f 2 acteur.tab  
Norris  
Stallone  
Seagal
```

- Order the actors in `acteur.tab` by (increasing) age

```
[stage11@nz ~]$ sort -n -k 3 acteur.tab  
Steven Seagal 61  
Sylvester Stallone 66  
Chuck Norris 72
```

- Replace the TAB character in `acteur.tab` with a semicolon (;). Store the result in `acteur.csv`

```
[stage11@nz ~]$ sed "s/\t/;/g" acteur.tab > acteur.csv
```


Using the `annuaire.csv` file

- Sort the file using the **team** column (6th)
- Extract the **name** (1st), **firstname** (2nd), **unit** (5th) and **team** (6th) columns
- Only keep people belonging to the **umr7144** unit.
- Store the result in file `annuaire_umr7144.csv`

All this using a single command line

TIMTOWDI

```
[stage11@nz ~]$ grep "umr7144" annuaire.csv | cut -d ";" -f "1,2,5,6" |  
sort -k 4 -t ";" > annuaire_umr7144.csv
```

```
[stage11@nz ~]$ sort -k 6 -t ";" annuaire.csv | grep "umr7144" | cut -d ";"  
-f "1,2,5,6" > annuaire_umr7144.csv
```

```
[stage11@nz ~]$ cut -d ";" -f "1,2,5,6" annuaire.csv | sort -k 4 -t ";" |  
grep "umr7144" > annuaire_umr7144.csv
```

Using the `condition2.go` file

- Determine the most frequent GO **number** (not the complete identifier, i.e. `0395853` in `GO:0395853`)

All this using a single command line

TIMTOWDI

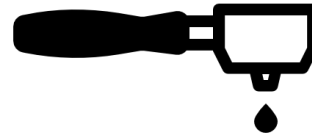
```
[stage11@nz ~]$ sort condition2.go | uniq -c | sort -k 1 -n | tail -1 | cut  
-f 1 | cut -f 2 -d ":"
```

```
[stage11@nz ~]$ sort condition2.go | uniq -c | sort -k 1 -r -n | head -1 |  
cut -f 1 | cut -f 2 -d ":"
```

- 1 A Quick Refresher
- 2 Redirections & Pipes
- 3 Slicing 'n Dicing files
- 4 Regular Expressions**
- 5 Awk 101
- 6 Batch files 101

Regular Expressions | A Definition

A regular expression, regex or regexp is, in **theoretical computer science** and **formal language** theory, a sequence of **characters** that define a search **pattern**. Usually this pattern is then used by **string searching algorithms** for "find" or "find and replace" operations on **strings**.



WIKIPEDIA
The Free Encyclopedia

A **sequence of characters** that define a **search pattern**

Two types of constraints define the pattern :

- **The very nature of the characters** : letters / digits / space or punctuation
- **The sequential organization of the characters** : the position(s) they are allowed to occupy in the sequence

Some real world examples :

- A (french domestic) phone number (i.e. 07 45 12 96 43) => a sequence of **5 groups** of **2 digits** each, separated by a **space character**.
- A DNA sequence coding for a (bacterial) protein => a series of **letters chosen from {a,t,g,c}** grouped by **triplets**, where the **first and last triplet** belong to two **specific subsets** of all possible triplets.

Regular Expressions | Character Classes

		grep	sed
[0-9]	Digits	✓	✓
[a-z]	Lowercase Letter	✓	✓
[A-Z]	Uppercase Letter	✓	✓
[a-zA-Z]	Alphabetic character	✓	✓
[0-9a-zA-Z]	Alphanumeric character	✓	✓
[\t_]	Space Character	✓	✓
.	Any character	✓	✓
[^ATGC]	Any character except ATGC	✓	✓

A sample pattern for a phone number :

[0-9]
[0-9]
[\t_]
[0-9]
[0-9]
[\t_]
[0-9]
[0-9]
[\t_]
[0-9]
[0-9]
[\t_]
[0-9]
[0-9]

Regular Expressions | Occurrences

		grep / sed -r	sed
?	Zero or one occurrence	✓	✗
+	At least one occurrence	✓	✗
*	Zero or more occurrences	✓	✓
{2}	Exactly two occurrences	✓	✗
{2,5}	From two to five occurrences	✓	✗
{2,}	At least two occurrences	✓	✗
{,5}	At most five occurrences	✓	✗

A sample pattern for a phone number including occurrence operators :

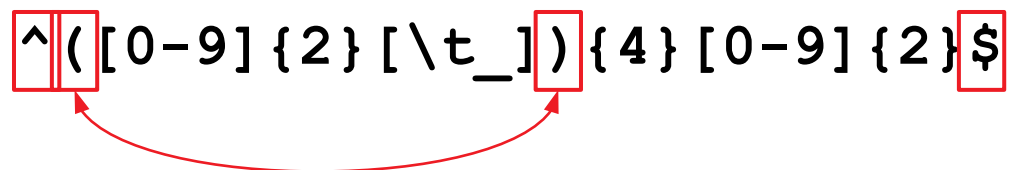
`[0-9]{2}[\t_][0-9]{2}[\t_][0-9]{2}[\t_][0-9]{2}[\t_][0-9]{2}`

Regular Expressions | Special characters

		grep / sed -r	sed
^	The beginning of a line	✓	✓
\$	The end of a line	✓	✓
	The “or” operator	✓	✗
(and)	The grouping operator	✓	✓
\	The “despecializing” character	✓	✓

A sample pattern for a single phone number on a line using grouping :

`^[([0-9]{2}[\t_]){4}[0-9]{2}$`



A sample pattern for search amounts in dollars with optional cents:

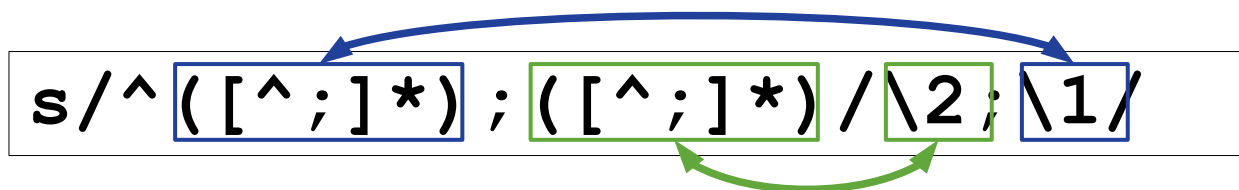
`\$[0-9]+(\.[0-9]+)?`

Regular Expressions | Using sed with REs

Patterns with regular expressions can be used when using **sed** for substitutions.

Each of the matches between parentheses can be referenced in the replacement string.

Ex.: swapping the first two columns in a CSV file using semi-colons



- `^` : anchor to the beginning of the line
- `[^;]*` : the contents of a field (any character except a semi-colon)
- `\1` : a reference to the first pattern between ()
- `\2` : a reference to the second pattern described between ()



When using **sed**, with regular expressions use option **-r**

Recommendation : use **egrep** (extended grep) instead of **grep**

egrep has better support for regular expressions

Ex.: Looking for phone numbers in the `annuaire.csv` file.

```
[stage11@nz ~]$ egrep --color "([0-9]{2} ){4}[0-9]{2}" annuaire.csv
Boye;Aurelien;aurelien.boye{AT}sb-roscoff.fr;02 98 29 23 23;fr2424;lbn
Czerwinska;Urszula;urszula.czerwinska{AT}sb-roscoff.fr;02 98 29 23 23;fr2424;lbn
Divoux;Jordane;jordane.divoux{AT}sb-roscoff.fr;02 98 29 23 23;fr2424;lbn
(...)
```

Using the `patelles_roscoff.tab` file

- Find all the pierced limpets (1 in the third column)

```
[stage11@nz ~]$ egrep --color "1$" patelles_roscoff.tab  
43,9      17,1      1  
42,8      15,8      1  
47,4      22,6      1  
(...)
```

Using the `annuaire.csv` file

- Find all the persons whose last name is Thomas

```
[stage11@nz ~]$ egrep --color "^Thomas;" annuaire.csv
Thomas;Wilfrid;wilfried.thomas{AT}sb-roscoff.fr;02 98 29 23 25;fr2424;service mer et
observation
Thomas;Serge;serge.thomas{AT}sb-roscoff.fr;02 98 29 23 48;umr7150;Physiologie
cellulaire
Thomas;Francois;francois.thomas{AT}sb-roscoff.fr;02 98 29 24 62;umr7139;Biochimie des
defenses chez les algues marines
Thomas;Mathilde;mathilde.thomas{AT}sb-roscoff.fr;02 98 29 23 23;fr2424;lbm
```

Using the `annuaire.csv` file

- Find all the persons whose first name is Thomas

```
[stage11@nz ~]$ egrep --color "^[^;]*;Thomas;" annuaire.csv  
Broquet;Thomas;thomas.broquet{AT}sb-roscoff.fr;02 98 29 23  
12;umr7144;Diversite et connectivite dans le paysage marin cotier
```

Using the `condition2.go` file

- Determine the most frequent GO **number** (not the complete identifier)

All this using a single command line including `sed` and a regular expression for the last stage

```
[stage11@nz ~]$ sort condition2.go | uniq -c | sort -k 1,1  
-n | tail -n 1 | sed -r "s/^. *GO: ([0-9]{7}) .*$/\1/"  
0003824
```

For the foolhearted : using the `nr.fsa` file

- Generate a two column file containing the access number (4th field of ID lines) and the organism name (between square brackets [])

```
[stage11@nz ~]$ grep ">" nr.fsa | sed -r "s/^>gi\|.*\|.*\|([A-Z]{2})_[0-9]*\.[0-9]*)\|.*\|(.*)\|.*$/\1\t\2/"
YP_005877138.1  Lactococcus lactis subsp. lactis IO-1
XP_642131.1  Dictyostelium discoideum AX4
XP_642837.1  Dictyostelium discoideum AX4
(...)
```


- 1 A Quick Refresher
- 2 Redirections & Pipes
- 3 Slicing 'n Dicing files
- 4 Regular Expressions
- 5 **Awk 101**
- 6 Batch files 101

AWK is a **pattern scanning** and **processing language**

pattern scanning : why bother, we already master **grep** and **sed** !

processing language : aren't we better off learning Python or R then ?

AWK fits in nicely for straightforward to moderately complex
line-oriented processing tasks.

- ✓ computations can be carried out on field values
- ✓ conditions can be checked before generating output
- ✓ programs can be stored in files for later reuse
- ✓ easy to use in pipe-based command-lines

The **awk** command-line is built as follows :

```
awk '{ instructions }' [file]
```

Where :

- **instructions** : recipe(s) describing operations to perform on the contents (substitute, delete, paste...)
- **file** : the file to act upon (optional : remember how pipes work ?)

- **awk** splits each input line in **fields** named **\$1**, **\$2**, **\$3** etc..
- The special **\$0** field **includes the whole line**.
- The **last field** of a line is stored in **\$NF**
- The **penultimate field** of a line is stored in **\$(NF-1)** etc...
- The **number of fields** of a line is stored in **NF** (**no dollar sign !**)
- The **current line number** in the input file is stored in **NR** (**no dollar sign !**)

The `print` instruction is used to generate output :

```
awk '{ print $0; }' [file]
```

Prints each line of the input stream to the output stream (!)

```
[stage11@nz ~]$ awk '{ print $2,$3 }' acteur.tab  
Norris 72  
Stallone 66  
Seagal 61  
(...)
```

Ex. : using `awk` to display the second and third columns of a file with added text.

```
[stage11@nz ~]$ awk '{ print $2" is "$3" years old." }' acteur.tab  
Norris is 72 years old.  
Stallone is 66 years old.  
Seagal is 61 years old.  
(...)
```

A **predicate** can determine if output will be generated for a given input line :

```
awk ' predicate { instructions }' [file]
```

Predicates most often verify **conditions** on one or more fields of the input line.

Predicates can use comparison operators :

- == (equality), and != (inequality)
- < (smaller), <= (smaller or equal), > (greater), >= (greater or equal)

Ex. : using **awk** to display veteran actors.

```
[stage11@nz ~]$ awk ' $NF >=65 { print $1" "$2 }' acteur.tab
```

```
Chuck Norris
```

```
Sylvester
```

```
(...)
```

Predicates can use regular expression operators :

- `~ /regexp/` : matches a regular expression
- `!~ /regexp/` : doesn't match a regular expression

Ex. : using `awk` with regular expressions to display actors whose name starts with "S"

```
[stage11@nz ~]$ awk ' $2 ~ /^S.* / { print $1 " " $2 }' acteur.tab  
Sylvester Stallone  
Steven Seagal  
(...)
```

Predicates can use arithmetic operators :

- `+`, `-`, `*`, `/`, `%`

Ex. : using `awk` with arithmetic operators to display actors with odd ages

```
[stage11@nz ~]$ awk ' $NF % 2 != 0 { print $0 }' acteur.tab  
Steven Seagal 61  
(...)
```

Predicates can use logical operators to combine terms:

- `term1 && term2` : true if both `term1` and `term2` evaluate as true
- `term1 || term2` : true if `term1` or `term2` (or both) evaluate as true

Ex. : using `awk` with arithmetic operators to display actors with even ages and who are over 70

```
[stage11@nz ~]$ awk ' $NF % 2 == 0 && $NF > 70 { print $0 }' acteur.tab  
Chuck Norris 72  
(...)
```

Two specially named blocks can be used to carry out instructions :

- Before the line processing loop : **BEGIN** block
- After all the lines have been processed **END** block

Ex. : using **BEGIN** to print output column headers

```
[stage11@nz ~]$ awk ' BEGIN { print "First Name\tLast Name\tAge" }  
{ print $0 }' acteur.tab  
First name      Last Name      Age  
Chuck Norris 72  
Sylvester Stallone 66  
(...)
```


Variables can be used in each block to store processing results.

Ex. : using variables to compute the average age of the actors.

```
[stage11@nz ~]$ awk ' BEGIN { total = 0 } { total=total+$3 } END { print "Average age "total/NR} ' acteur.tab
Average age 66.3333
(...)
```

Some functions that can be used with variables :

- **length(s)** : number of characters in **s**
- **toupper(s)** : transform **s** to uppercase letters
- **tolower(s)** : transform **s** to lowercase letters
- **sub(r,s,t)** : replace every match of regexp **r** with string **s** in **t**
- **split(s,a,d)** : split string **s** using delimiter **d** and store the result in array **a**

- **int(n)** : compute the integer part of **n**
- **log(n)** : compute the logarithm part of **n**
- **sqrt(n)** : compute the square root of **n**

Specifying the field delimiter

```
awk -F ';' 'predicate { instructions }' [file]
```

Using awk with a file containing the instructions :

```
awk -f myawkprogram.txt [file]
```

Using the `patelles_roscoff.tab` file

- Find all the pierced limpets :
 - using `awk` with an arithmetic operator predicate
 - using `awk` with a regular expression operator predicate

```
[stage11@nz ~]$ awk '$NF == 1 { print $0}' patelles_roscoff.tab
43,9      17,1      1
42,8      15,8      1
47,4      22,6      1
(...)
```

```
[stage11@nz ~]$ awk '$0 ~ /^.*1$/ { print $0}' patelles_roscoff.tab
43,9      17,1      1
42,8      15,8      1
47,4      22,6      1
(...)
```

Using the `annuaire.csv` file

- Find all the persons whose last name is Thomas (using `awk` obviously)

```
[stage11@nz ~]$ awk -F ';' ' $1 == "Thomas" { print $0}' annuaire.csv
Thomas;Wilfrid;wilfried.thomas{AT}sb-roscoff.fr;02 98 29 23 25;fr2424;service mer et
observation
Thomas;Serge;serge.thomas{AT}sb-roscoff.fr;02 98 29 23 48;umr7150;Physiologie
cellulaire
(...)
```

- 1 A Quick Refresher
- 2 Redirections & Pipes
- 3 Slicing 'n Dicing files
- 4 Regular Expressions
- 5 Awk 101
- 6 Batch files 101**

What's a batch file ?

- **Level 0** : A text file with a series of commands
- **Level 1** : Level 0 + with input parameters to configure the command execution
- **Level 2** : Level 1 + control structures (conditionals, loops)
- **Level 3** : Level 2 + functions

Why use batch files ?

- **Level 0** : To avoid tediously retyping complex commands
- **Level 1** : To reuse series of commands with different parameter sets
- **Level 2** : To make batch execution more robust
- **Level 3** : Because for command-line based tasks it beats programming languages

Your most basic batch file

Ex. : writing a batch file to display the most recent files in directory `/tmp`

```
1 [stage11@nz ~]$ cat > ./tmpsmostrecent.sh
2 ls -lat | head -5
  [Ctrl^D]
3 [stage11@nz ~]$ chmod +x ./tmpsmostrecent.sh
4 [stage11@nz ~]$ ./tmpsmostrecent.sh
total 104
drwxrwxrwt. 12 root      root      57344 Nov 26 14:25 .
drwxr-xr-x  2 stage11   stage     4096  Nov 26 14:25 1279165.1.qlogin.q
(...)
```

- 1 Create an empty file using **cat** and **stdin**
- 2 Type the commands you want in your batch file and finish with **[Ctrl^D]**
- 3 Make the file executable using **chmod**
- 4 Run your batch file

Using Batch Files | Choosing the Shell

Rationale : leaving the choice of the shell to the system might (sometimes) lead to *minor incompatibilities* when *copying batch files* to other environments.

Rule of thumb : always start your batch scripts with the following line :

```
#!/usr/bin/env bash
```

#! (or she-bang) : tells the system that your file is a (batch) script needing an interpreter

/usr/bin/env bash tells the system the interpreter is the bash version configured in your environment

Using Batch Files | Argument Passing

The canonical command-line structure also applies to batch files

```
./mybatch.sh arg1 arg2 arg3...argn
```

```
#!/usr/bin/env bash

# hello.bash :
# A simple batch file writing its first argument to stdout

echo "Hello $1"
```

```
[stage11@nz ~]$ ./hello.bash Guru
Hello Guru
```

The special variable **\$0** matches the command name (i.e the name of the batch file)

The special variable **\$*** matches the whole set of arguments

- Write a batch file listing the most recent files of a directory.
- The name of the directory and the number of files to be displayed are passed as arguments to the batch file.

```
#!/usr/bin/env bash

# mostrecent.bash :
# A simple batch file displaying the most recent files in
# a directory
# Usage : mostrecent.bash directoryname numberofiles

ls -lat $1 | head -n $2
```

Using Batch Files | Basic Loops

The loop structure is used to apply a series of commands to a sequence of words :

```
for <word> in <wordlist> ; do  
  
    # use ${<word>} in various commands  
  
done
```

```
#!/usr/bin/env bash  
  
# dispargs.bash :  
# A simple batch file using the for loop to enumerate its  
# arguments  
  
for userarg in $* ; do  
    echo "The next argument is ${userarg}"  
done
```

```
[stage11@nz ~]$ ./dispargs.bash Gnu is Not Unix  
The next argument is Gnu  
The next argument is is  
The next argument is Not  
The next argument is Unix
```

Using Batch Files | Looping over Files

A frequent use case of loops is to apply a series of commands on files in a directory, relying on `ls` to retrieve the file list as in :

```
files=$(ls <directory>)

for file in ${files} ; do

    # use ${file} for useful stuff

done
```

The `$(<commands>)` construction, runs the `<commands>` and returns what they write to `stdout`

```
#!/usr/bin/env bash

# protect.bash :
# A simple batch file to write-protect the contents of a directory
# given as argument argument

files=$(ls $1)

for file in ${files} ; do
    chmod -w ${file}
done
```

- Write a batch file taking a file extension and directory name as arguments and displaying : the owner, the size and the filename.

```
#!/usr/bin/env bash

# customls.bash :
# A simple batch file displaying some info about files
# with a given extension in a specific directory
# Usage : customls.bash extension directoryname

files=$(ls $2/*. $1)
for file in ${files} ; do
    ls -l ${file} | awk '{print $3,$5,$NF}'
done
```

- Write a batch file using `annuaire.csv` to generate a file with two columns : the name of the lab and the number of members of the lab.
- Hint : generate temporary files with the members of each lab.

```
#!/usr/bin/env bash

# catcount.bash :
# A batch file generating a file with the number lines for
# each different value of a given column in the input file.

# Usage : catcount.bash inputcsvfile categorycolumn outputfile

csvfile=$1
rm -f /tmp/categories_*/csv
# Stage 1 : generate the intermediate files for each category
for line in $(sed 's/ /#/g' ${csvfile}) ; do
    lab=$(echo ${line} | cut -d ";" -f $2)
    echo ${line} | sed 's/#/ /g' >> /tmp/categories_${lab}.csv
done

# Stage 2 : count the lines in the intermediate files
rm -f $2
for labfile in $(ls /tmp/categories_*.csv) ; do
    lab=$(echo ${labfile} | sed -r 's/^.*_([a-z0-9]+)\.csv$/\1/')
    members=$(wc -l ${labfile} | awk '{print $1}')
    echo "${lab} ${members}" >> $3
done
```



Thank you for your patience and your tenacity