Advanced Linux

ABiMS Training Module 2022

Gildas Le Corguillé

Credit: Philippe Bordron, Mark Hoebeke, Gildas Le Corguillé

Advanced Linux | Goals

- 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 ?

CNRS • SORBONNE UNIVERSITÉ
Station Biologique de Roscoff
ABiMS

1. A Quick Refresher
2. Redirections & Pipes
3. Slicing ‘n Dicing Files
4. Regular Expressions
5. Awk 101
6. Batch Scripts 101

OUTLINE

CNRS • SORBONNE UNIVERSITÉ
Station Biologique de Roscoff
ABiMS

1. A Quick Refresher
2. Redirections & Pipes
3. Slicing ‘n Dicing Files
4. Regular Expressions
5. Awk 101
6. Batch Scripts 101

A QUICK REFRESHER

Quick Refresher | CLI Survival Kit

Where am I ?

CNRS • SORBONNE UNIVERSITÉ
Station Biologique de Roscoff
ABiMS

Quick Refresher | CLI Survival Kit

Where am I ?

```
[stage01@slurm0 ~]$ pwd
```

Which files/directories are located “here” ?

Quick Refresher | CLI Survival Kit

Where am I ?

```
[stage01@slurm0 ~]$ pwd
```

Which files/directories are located "here" ?

```
[stage01@slurm0 ~]$ ls
[stage01@slurm0 ~]$ ls .
```

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

Quick Refresher | CLI Survival Kit

Where am I ?

```
[stage01@slurm0 ~]$ pwd
```

Which files/directories are located "here" ?

```
[stage01@slurm0 ~]$ ls
[stage01@slurm0 ~]$ ls .
```

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

```
[stage01@slurm0 ~]$ ls -la /tmp
```

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

Quick Refresher | CLI Survival Kit

Where am I ?

```
[stage01@slurm0 ~]$ pwd
```

Which files/directories are located “here” ?

```
[stage01@slurm0 ~]$ ls
[stage01@slurm0 ~]$ ls .
```

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

```
[stage01@slurm0 ~]$ ls -la /tmp
```

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

```
[stage01@slurm0 ~]$ cd /tmp
```

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

CNRS • SORBONNE UNIVERSITÉ
Station Biologique de Roscoff
ABiMS

Quick Refresher | CLI Survival Kit

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

```
$ mkdir ~/foo/bar
```

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

Quick Refresher | CLI Survival Kit

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

```
$ mkdir ~/foo/bar
```

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

```
$ cp truc ~/foo/bar
```

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

Quick Refresher | CLI Survival Kit

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

```
$ mkdir ~/foo/bar
```

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

```
$ cp truc ~/foo/bar
```

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

```
$ mv truc ~/foo/bar
```

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

Quick Refresher | CLI Survival Kit

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

```
$ mkdir ~/foo/bar
```

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

```
$ cp truc ~/foo/bar
```

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

```
$ mv truc ~/foo/bar
```

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

```
$ cp -r trucs/ ~/foo/bar
```

Quick Refresher | CLI Survival Kit

How do I remove (delete forever) file `truc` ?

Quick Refresher | CLI Survival Kit

How do I remove (delete forever) file `truc` ?

```
$ rm truc
```

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

Quick Refresher | CLI Survival Kit

How do I remove (delete forever) file `truc` ?

```
$ rm truc
```

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

```
$ rm -rf truc
```

How do I remove *empty* directory `truc` ?

Quick Refresher | CLI Survival Kit

How do I remove (delete forever) file `truc` ?

```
$ rm truc
```

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

```
$ rm -rf truc
```

How do I remove *empty* directory `truc` ?

```
$ rmdir truc
```

Quick Refresher | CLI Survival Kit

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

Quick Refresher | CLI Survival Kit

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

```
$ file truc
```

How do I display the contents of file `truc` (and recover control of the terminal right away) ?

Quick Refresher | CLI Survival Kit

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

```
$ file truc
```

How do I display the contents of file `truc` (and recover control of the terminal right away) ?

```
$ cat truc
```

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

Quick Refresher | CLI Survival Kit

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

```
$ file truc
```

How do I display the contents of file `truc` (and recover control of the terminal right away) ?

```
$ cat truc
```

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

```
$ head truc
$ tail truc
```

How do I page through file `truc` ?

Quick Refresher | CLI Survival Kit

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

```
$ file truc
```

How do I display the contents of file `truc` (and recover control of the terminal right away) ?

```
$ cat truc
```

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

```
$ head truc
$ tail truc
```

How do I page through file `truc` ?

```
$ less truc
```

How do I edit file `truc` ?

Quick Refresher | CLI Survival Kit

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

```
$ file truc
```

How do I display the contents of file `truc` (and recover control of the terminal right away) ?

```
$ cat truc
```

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

```
$ head truc
$ tail truc
```

How do I page through file `truc` ?

```
$ less truc
```

How do I edit file `truc` ?

```
$ gedit truc
```

Quick Refresher | CLI Survival Kit

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

Quick Refresher | CLI Survival Kit

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

```
$ matrix &
```

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

Quick Refresher | CLI Survival Kit

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

```
$ matrix &
```

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

```
$ matrix
[Ctrl+Z]
(…)
$ bg
```

When in doubt about running a program :

```
$ man command
```

```
$ command --help
$ command -h
$ command -help
$ command help
$ command sub --help
$ command
```

Connexion to the cluster

1. Open a terminal and connect to slurm0

```
[cnorris@desktop ~]$ ssh -Y stage01@slurm0.sb-roscoff.fr
```

2. Jump to one of the cluster nodes **(nobody runs jobs on slurm0 !)**

```
[stage01@slurm0 ~]$ srun --pty bash
```

- `srun` slurm command to run a job on one node
- `--pty` Execute task zero in pseudo terminal mode
- `bash` The program is bash because we will do bash

3. Go to your “project” directory **(don’t work in you home directory !)**

```
[stage01@n118~]$ cd  /shared/projects/stage/stageXX
```

CNRS • SORBONNE UNIVERSITÉ Station Biologique de Roscoff ABiMS

Connexion to the cluster

4. Get the course material

```
[stage01@n118 stageXX]$ wget https://frama.link/Linux-Avance
```

5. Unpack the course material

```
[stage01@n118 stageXX]$ tar -zxvf Linux-Avance
```

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

REDIRECTIONS & PIPES

Redirections & Pipes

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

- 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

```
$ ls -l * > listoffiles.txt
$ cat listoffiles.txt
-rwxr-xr-x 1 stage01 stage        55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 stage01 stage       488 sept.  5  2012 address.tab
-rwxr-xr-x 1 stage01 stage     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

- 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

```
$ grep tab < listoffiles.txt
-rwxr-xr-x 1 stage01 stage       55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 stage01 stage      488 sept.  5  2012 address.tab
-rwxr-xr-x 1 stage01 stage  1315419 sept.  5  2012 insulin.vs.nt.blastn.tab
(...)

$ wc -l insulin.vs.nt.blastn.tab
12172 insulin.vs.nt.blastn.tab
$ wc -l < insulin.vs.nt.blastn.tab
12172
```

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

- 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

```
$ grep tab < listoffiles.txt > tabfiles.txt
$ cat tabfiles.txt
-rwxr-xr-x 1 stage01 stage       55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 stage01 stage      488 sept.  5  2012 address.tab
-rwxr-xr-x 1 stage01 stage  1315419 sept.  5  2012 insulin.vs.nt.blastn.tab
(...)
```

Redirections & 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**

```
$ ls -l | grep tab
-rwxr-xr-x 1 stage01 stage       55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 stage01 stage      488 sept.  5  2012 address.tab
-rwxr-xr-x 1 stage01 stage  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.

```
$ ls -l | grep tab | grep -v insulin
-rwxr-xr-x 1 stage01 stage       55 sept.  5  2012 acteur.tab
-rwxr-xr-x 1 stage01 stage      488 sept.  5  2012 address.tab
(...)
```

Redirections & Pipes

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.

```
$ ls -l /home/fr2424/stage/* > /tmp/lsfr2424_11.txt
ls: cannot open directory /home/fr2424/stage/stage01: Permission denied
ls: cannot open directory /home/fr2424/stage/stage02: Permission denied
ls: cannot open directory /home/fr2424/stage/stage03: Permission denied
(...)
```

The redirection of **stderr** is possible by adding the **2>** redirection symbol after a command's arguments.

```
$ ls -l /home/fr2424/stage/* > /tmp/lsfr2424_11.txt 2> /tmp/lserrors_11.txt
$ cat /tmp/lserrors.txt
ls: cannot open directory /home/fr2424/stage/stage01: Permission denied
ls: cannot open directory /home/fr2424/stage/stage02: Permission denied
ls: cannot open directory /home/fr2424/stage/stage03: Permission denied
(...)
```

Redirections & Pipes

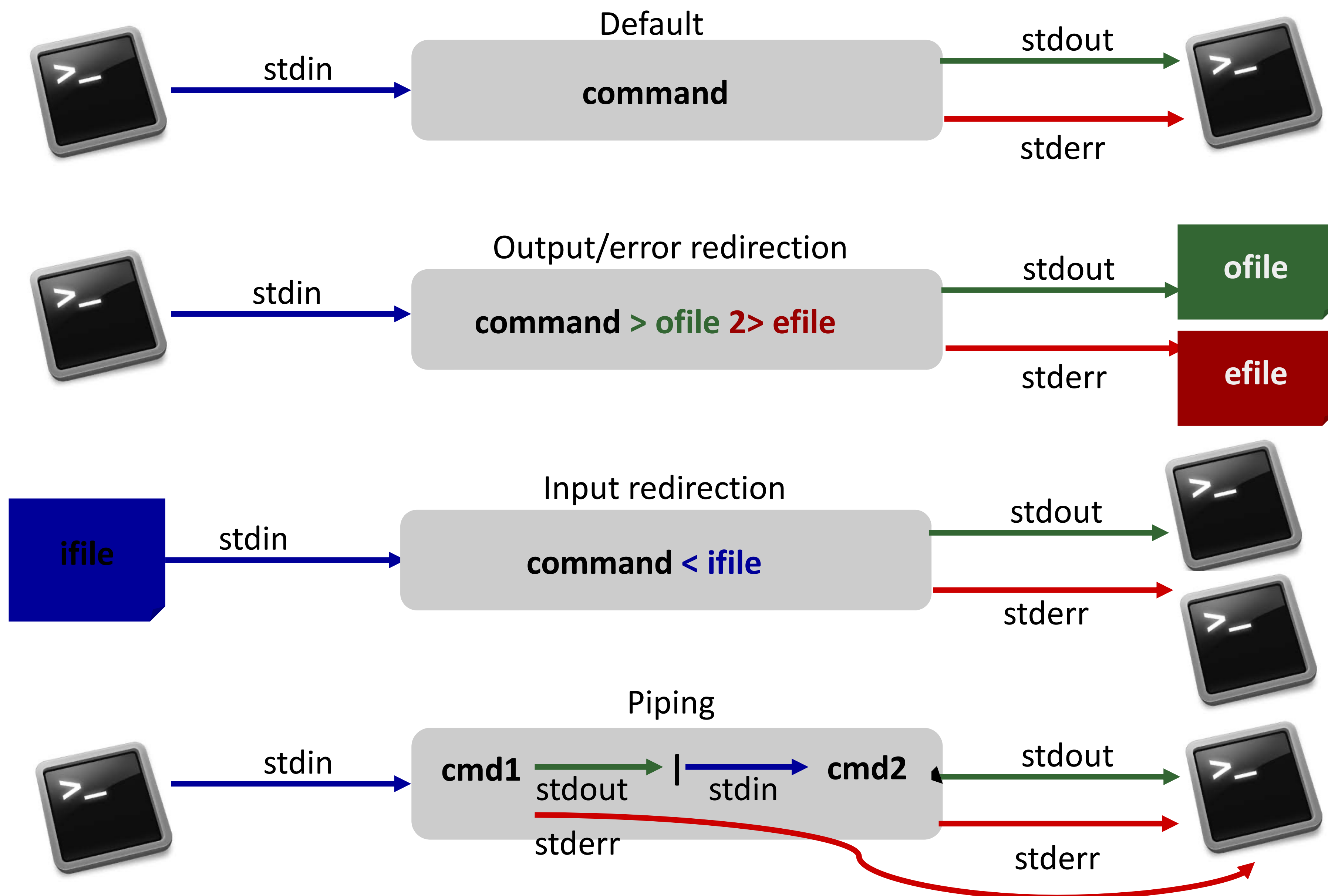
- To ignore what's generated on an output channel (stdout or stderr), it can be redirected to a special file : /dev/null.

```
$ ls -lR /home/fr2424 > /tmp/lsfr2424.txt 2> /dev/null
$
```

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

```
$ ls *.tab > fileswithcolumns.txt
$ wc -l fileswithcolumns.txt
5 fileswithcolumns.txt
$ ls *.csv >> fileswithcolumns.txt
$ wc -l fileswithcolumns.txt
7 fileswithcolumns.txt
```

Redirections & Pipes



1. A Quick Refresher
2. Redirections & Pipes
3. Slicing ‘n Dicing Files
4. Regular Expressions
5. Awk 101
6. Batch Scripts 101

SLICING ‘N DICING FILES

Slicing 'n Dicing

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:

```
$ ./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:

```
$ dos2unix ./phyml-mpi-multi.sh
```

Slicing 'n Dicing

grep

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

```
$ 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).

```
$ grep -r -c -i TRANSCRIPT .
./insulin_vs_nt.blast:144
./acteur.csv:0
./insulin.fas:5
```

Slicing 'n Dicing

grep

When the relevant information spans several lines, **grep** can give contextual information

- **-A** ***n*** option, **grep** displays for each matching line, the ***n*** **following** lines (A: after)

```
$ 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
TTTTGTAGATAAATGTGAGGATTTTCTCTAAATCCCTCTTCTGTTTGCTAAATCTCACTGTCACTGCTAA
--
>gi|163659900|ref|NM_001111284.1| Homo sapiens insulin-like growth factor 1 (somatomedin C) (IGF1), transcript variant 2, mRNA
GCATACCTGCCTGGGTGTCCAAATGTAACTAGATGCTTTCACAAACCCCACCCACAAAGCAGCACATGTT
--
(...)
```

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

```
$ 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

cut

The command takes an option describing how to extract columns (aka fields) and an argument with the name of the file

```
$ cut -f 1 acteur.tab
Chuck
Sylvester
Steven
(...)
$ cut -f "2,3" acteur.tab          # have you try -f "3,2"?
Norris    72
Stallone  66
Seagal    61
(...)
$ cut --complement -f "2,3" acteur.tab
Chuck
Sylvester
Steven
(...)
$ cut -d ";" -f 2 annuaire.csv
Clio
Brice
Mathilde
(...)
```

Slicing 'n Dicing

sort

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

```
$ sort pop_ville.tab
Paris      4193031
Roscoff    3705
Tokyo      13010279
$ sort -k 2 pop_ville.tab
Tokyo      13010279
Roscoff    3705
Paris      4193031
$ sort -n -k 2 pop_ville.tab
Roscoff    3705
Paris      4193031
Tokyo      13010279
$ sort -r -n -k 2 pop_ville.tab
Tokyo      13010279
Paris      4193031
Roscoff    3705
```

uniq

The command is used to remove consecutive identical lines in a file.

On a sorted file, it removes all repeated lines.

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

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

```
$ uniq -c condition1_sorted.go
      2 GO:0000166  nucleotide binding
      1 GO:0003824  catalytic activity
      7 GO:0005488  binding
(...)
```

- **-u** extract unique occurrences

```
$ uniq -u condition1_sorted.go
GO:0003824          catalytic activity
GO:0005623          cell
GO:0006810          transport
GO:0008152          metabolic process
```

Slicing 'n Dicing

uniq

The command is used to remove consecutive identical lines in a file.

On a sorted file, it removes all repeated lines.

```
$ uniq -c condition1.go
      2 GO:0016787  hydrolase activity
      1 GO:0005623  cell
      1 GO:0030154  cell differentiation
      1 GO:0005488  binding
      1 GO:0016787  hydrolase activity
      1 GO:0005737  cytoplasm

$ sort condition1.go > condition1_sorted.go
$ uniq -c condition1_sorted.go
      6 GO:0000166  nucleotide binding
      2 GO:0003677  DNA binding
     12 GO:0003824  catalytic activity
      3 GO:0004672  protein kinase activity
      2 GO:0005215  transporter activity


$ uniq -c <(sort condition1.go)
      6 GO:0000166  nucleotide binding
      2 GO:0003677  DNA binding
     12 GO:0003824  catalytic activity
      3 GO:0004672  protein kinase activity
      2 GO:0005215  transporter activity
```

join

The **join** command is used to 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**

```
$ head -1 acteur_sorted.tab
Chuck   Norris  72
$ head -1 address_sorted.tab
Canet   Guillaume       Artmedia 20, Avenue Rapp 75007 Paris  France
$ 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

join

The **`join`** command is used to 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`**

```
$ 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


$ join -i -1 2 -2 1 <(sort -k 2 acteur.tab) <(sort -k 1 address.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

sed

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

```
$ 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

sed

Ex.: Field delimiter substitution

```
$ 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

Slicing 'n Dicing

sed

Ex.: Using locations to operate on specific line ranges

```
$ 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

```
$ sed "2d" acteur.tab
Chuck   Norris  72
Steven  Seagal  61
```

Ex.: Combining operators: pasting & replacing

```
$ sed "2p; s/Sylvester/Sly/" acteur.tab
Chuck   Norris  72
Sylvester       Stallone66
Sly     Stallone66
Steven  Seagal  61
```

Exercises

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

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

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

CNRS • SORBONNE UNIVERSITÉ Station Biologique de Roscoff ABiMS

Exercises

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

```
$ cut -f 2 acteur.tab
Norris
Stallone
Seagal
```

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

```
$ sort  -n -k 3 acteur.tab
Steven   Seagal   61
Sylvester         Stallone66
Chuck    Norris   72
```

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

```
$ sed "s/\t/;/g" acteur.tab > acteur.csv
```

Exercises

Using the **`annuaire.csv`** file

- Sort the file using the **team** column (6)

- Extract the **name** (1), **firstname** (2), **unit** (5) and **team** (6) columns

- Only keep people belonging to the **umr7144** unit.

- Store the result in file **`annuaire_umr7144.csv`**

All this using a single command line

Exercises

TIMTOWDI

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

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

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

Exercises

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

Exercises

TIMTOWDI

```
$ sort condition2.go | uniq -c | sort -k 1 -n | tail -1 | cut -f 1 | cut -f 2 -d ":"
```

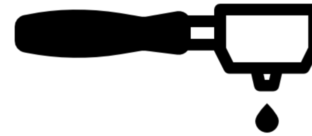
```
$ sort condition2.go | uniq -c | sort -k 1 -r -n | head -1 | cut -f 1 | cut -f 2 -d ":"
```

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

REGULAR EXPRESSIONS

Regular Expressions

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.



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 to search for 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

```
s/^([^;]*);([^;]*)/\2;\1/
```

`^` : 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`

Regular Expressions | Using grep with REs

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.

```
$ 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;lbm
Czerwinska;Urszula;urszula.czerwinska{AT}sb-roscoff.fr;02 98 29 23 23;fr2424;lbm
Divoux;Jordane;jordane.divoux{AT}sb-roscoff.fr;02 98 29 23 23;fr2424;lbm
(...)
```

Exercises

Using the `patelles_roscoff.csv` file

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

Exercises

Using the `patelles_roscoff.csv` file

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

```
$ egrep --color "1$" patelles_roscoff.csv
43,9            17,1            1
42,8            15,8            1
47,4            22,6            1
(...)
```

More secure

```
$ grep -P "^([0-9,]+\t){2}1" patelles_roscoff.csv
43,9            17,1            1
42,8            15,8            1
47,4            22,6            1
(...)
```

Exercises

Using the `annuaire.csv` file

- Find all the persons whose last name is Thomas

Exercises

Using the `annuaire.csv` file

- Find all the persons whose last name is Thomas

```
$ 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
```

Exercises

Using the `annuaire.csv` file

- Find all the persons whose first name is Thomas

Exercises

Using the **annuaire.csv** file

- Find all the persons whose first name is Thomas

```
$ 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
```

```
$ egrep --color "^[A-Z][a-z]+;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
```

Exercises

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

```
$ sort condition2.go | uniq -c | sort -k 1 -n | tail -1 | cut -f 1 | cut -f 2 -d ":"
```

```
$ sort condition2.go | uniq -c | sort -k 1 -n | tail -1 | sed  (...)
```

Exercises

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

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

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

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

Exercises

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

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

```
$ 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
(…)
```

Station Biologique de Roscoff
CNRS • SORBONNE UNIVERSITÉ
ABiMS

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

AWK 101

Awk 101 | What's Awk ?

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**

Awk 101 | Basic Command-Line Structure

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 !)**

Awk 101 | Instructions

The **`print`** instruction is used to generate output:

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

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

```
$ 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.

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

Awk 101 | Conditional Output

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.

```
$ awk '$3 >=65 { print $1" "$2 }' acteur.tab
Chuck Norris
Sylvester Stallone
(...)
```

Awk 101 | Conditional Output

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"

```
$ 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

```
$ awk ' $3 % 2 != 0 { print $0 }' acteur.tab
Steven Seagal 61
(...)
```

Awk 101 | Conditional Output

Predicates can use logical operators to combine terms:

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

Ex.: using **awk** with arithmetic operators to display actors whose name starts with “S” and who are over 65

```
$ awk ' $2 ~ /^S.*/  && $3 > 65 { print $0 }' acteur.tab
Sylvester     Stallone      66
(...)
```

Station Biologique de Roscoff — CNRS • Sorbonne Université — ABiMS

Awk 101 | BEGIN and END blocks

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

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

Awk 101 | Using Variables

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

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

```
$ 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**

```
$ awk '{ total+=$3 } END { print total/NR} ' acteur.tab
66.3333
```

Awk 101 | Tips

Specifying the field delimiter

```
awk -F ';' 'predicate { instructions }' [file]
awk 'BEGIN { FS="\t" } predicate { instructions }' [file]
```

Specifying the output field delimiter

```
awk 'BEGIN { OFS="\t" } predicate { instructions }' [file]
```

Using awk with a file containing the instructions:

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

Exercises

Using the `patelles_roscoff.csv` file

- Find all the pierced limpets:
  - using **`awk`** with an arithmetic operator predicate

Exercises

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

```
$ awk '$3 == 1 { print $0}' patelles_roscoff.tab
43,9        17,1        1
42,8        15,8        1
47,4        22,6        1
(...)
```

Exercises

Using the **`annuaire.csv`** file

- Find all the persons whose first name is Thomas (using **`awk`**)

Exercises

Using the **annuaire.csv** file

- Find all the persons whose first name is Thomas (using **awk**)

```
$ awk -F ';' ' $2 == "Thomas" { print $0}' annuaire.csv
Broquet;Thomas;thomas.broquet{AT}sb-roscoff.fr;02 98 29 23 12;umr7144;Diversite et
connectivite dans le paysage marin cotier

$ awk -F ';' ' $2 ~ /^Thomas$/ { print $0}' annuaire.csv
Broquet;Thomas;thomas.broquet{AT}sb-roscoff.fr;02 98 29 23 12;umr7144;Diversite et
connectivite dans le paysage marin cotier
```

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

BATCH SCRIPTS 101

Batch Scripts 101 | Why?

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

Batch Scripts 101 | How?

Your most basic batch file

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

```
$ gedit ./tmpsmostrecent.sh &
```

```
ls -t | head -5
```

```
$ chmod +x ./tmpsmostrecent.sh  # Make the file executable using chmod

$ ./tmpsmostrecent.sh           # Run your batch file
acteur.tab
condition2.go
hmmpfam.out
spur_transcriptome.fna
iprscan.xml
```

Batch Scripts 101 | Choosing an interpreter

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

Batch Scripts 101 | Passing Arguments

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"
```

```
$ ./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

Exercises

- 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.

Exercises

- 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 -t $1 | head -n $2
```

Batch Scripts 101 | Basic loop

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
```

```
$ ./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
```

Batch Scripts 101 | 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

# head.bash:
# A simple batch file to display the 10th first line of the files
# within a directory given as argument argument

files=$(ls $1)

for file in ${files} ; do
        head ${file}
done
```

Batch Scripts 101 | Looping over files

Loop are also possible within the terminal for little but useful tasks

```
$ for i in $(seq -w 1 36); do mkdir stage${i}; done
$ ls
stage01  stage05  stage09  stage13  stage17  stage21  stage25  stage29  stage33
stage02  stage06  stage10  stage14  stage18  stage22  stage26  stage30  stage34
stage03  stage07  stage11  stage15  stage19  stage23  stage27  stage31  stage35
stage04  stage08  stage12  stage16  stage20  stage24  stage28  stage32  stage36
```

Exercises

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

Exercises

- 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
```

ABiMS

Exercises

- Write a batch file which changes the extension of your file .tab to .tsv

Dataset:

```
$ for file in $(seq -w 1 100); do touch $file.tab; done
```

- Level 2: the 2 extensions are some parameters
- Level 3: duplicate the files as well as changing their extension in another directory

```
$ ./changeext tab tsv . results/
```

Exercises

- Write a batch file which change the extension of your file .tab to .tsv

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

# changeext.sh:
# This bash script allow to change some files extensions

for FILE in $(ls *.tab) ; do
      mv $FILE $(echo $FILE | sed "s|\.tab$|\.tsv|")
done
```

Exercises

- Write a batch file which change the extension of your file .tab to .tsv
- Level 2: the 2 extensions are some parameters

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

# changeext.sh ext1 ext2:
# This bash script allow to change some files extensions
EXT1=$1
EXT2=$2


for FILE in $(ls /*.$EXT1) ; do
	mv $FILE $(echo $FILE | sed "s|\.$EXT1$|\.$EXT2|")
done
```

Exercises

- Write a batch file which change the extension of your file .tab to .tsv
- Level 3: duplicate the files as well as changing their extension in another directory

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

# changeext.sh tab tsv . results/
# This bash script allow to change some files extensions
EXT1=$1
EXT2=$2
DIR1=$3
DIR2=$4
mkdir $DIR2 2> /dev/null
for FILE in $(ls $DIR1/*.$EXT1) ; do
       cp $FILE $DIR2/$(basename $FILE | sed "s|\.$EXT1|\.$EXT2|")
done
```

Thank you for your patience and your tenacity