

# Linux -SHELL :

**PROGRAMMATION - les bases**

**Thibault Marchal**

# Sommaire 1/4

- Introduction et Shell
  - Le rôle du shell
  - Les différents shells
  - Les alias
  - Interprétation d'une commande
- L'exécution d'un script et débogage
  - Les méthodes d'exécution
  - Le shebang
  - Les commentaires
  - La structure d'un script
  - Le débogage
- Les variables
  - Les variables
  - Les manipulations avancées
  - La concaténation
  - L'isolation
  - Substitution

# Sommaire 2/4

- L'interactivité avec un script
  - La commande read
  - Le passage d'argument
  - Le mot clé return
  - L'externalisation des fonctions
- Les tests, les opérateurs if et case
  - Le code de retour \$ ?
  - Les opérateurs && et ||
  - La commande test
  - L'utilisation des tests conditionnels if et case
- Les boucles
  - Les boucles for, while et until
  - Les instructions break, continue et exit

# Sommaire 3/4

- Le traitement arithmétique
  - Les instructions `expr`, `let` et `bc`
  - L'utilisation de `(( ))`
- Le traitement des chaînes de caractères
  - Typé une variable
  - Manipulation des chaînes de caractères
- Les fonctions
  - La déclaration
  - Le passage d'argument
  - Le mot clé `return`
  - L'externalisation des fonctions

# Sommaire 4/4

- Les expressions régulières et les commandes grep
  - Les expressions régulières
  - Les commandes grep, fgrep et egrep
- La commande sed
  - La syntaxe, son potentiel
  - Quelques cas
- La commande awk
  - La syntaxe
  - Les opérations

# Introduction et Shell

- Le rôle du shell
- Les différents shell
- Les alias
- Terminal virtuel

# Le rôle du shell

**Rappel :**

**Le shell est l'intermédiaire entre l'os et l'utilisateur**  
**C'est un programme**

# Les différents shells

- Il en existe plusieurs.
  - Celui par default que vous utilisiez bash appelé «Bourne Again Shell »
  - Sh «Bourne Shell »
  - Csh « C shell »
  - Tsch «Texnec C shell »
  - Ksh « Korn shell »
  - Zsh « zero shell »
- Pour voir celui par default d'un user «cat /etc/passwd », *c'est le dernier champ*

```
marchal:x:1000:1000:marchal,,,:/home/marchal:/bin/bash
```



# Les alias

- Un alias c'est un raccourci qui fait référence a une commande
  - Cela sert surtout à remplacer des commandes longues et récurrentes par une version plus courtes, ex :
    - Ls -al raccourci en la
    - gssh : ssh -6 `nom_d_utilisater@20012a00`[ipv6]
- Pour ajouter un alias :
- Depuis votre terminal (temporaire):
  - `alias nom_de_l_alias='votre_commande_entre_quote'`
    - Ex : `alias la='ls -la'`
  - Modifier le fichier `.bashrc` ou `.bash_aliases` sur certaines distrib, ajouter a la fin du fichier (permanent):
    - `alias la='ls -la'`

# Terminal virtuel

- Plusieurs terminaux virtuels (`/dev/tty*`) sont lancés automatiquement lors du boot
- Vous pouvez y accéder via le raccourci `<ctrl> + <alt> + <F1-6>` (Fn correspond aux touches de fonction numérotées du clavier)
- Source et plus de détails:  
[https://linuxpedia.fr/doku.php/commande/interpreteur\\_de\\_commandes](https://linuxpedia.fr/doku.php/commande/interpreteur_de_commandes)

# L'exécution d'un script et débogage

Les méthodes d'exécution

Le shebang

Les commentaires.

La structure d'un script et les bonnes pratiques

Le débogage.

# Les méthodes d'exécution

- En graphique, en général clic droit sur un .sh → propriétés → permissions → coché case « Execution »
- Via le terminal,
  - `bash <nom_de_votre_script>`
  - `./<nom_de_votre_script>`
- En modifiant le PATH (variables environment)
  - Dans votre fichier .bashrc ajouter à PATH : «`:/<dossier>/<qui>/<va>/<à>/<votre>/<script>`»

# Le shebang

- Le shebang commence toujours « #! »
- Il est suivi par le chemin de votre interpréteur
  - Ex :
    - #! /bin/bash (script bash)
    - #! /usr/bin/python (script pytho,)

–

# Les commentaires

- Chaque langage possède un symbole pour déclarer un commentaire.
  - Sur linux en général c'est «# »
  - L'interpréteur choisie n'essayera pas d'interprété cette ligne
- Les commentaires sont voués à rendre votre code compréhensible par d'autres utilisateurs
  - Détaillé toujours ce que vous voulez faire

# La structure d'un scripte et bonne pratique

- Le shebang toujours en première ligne
- Accompagné de quelque ligne commenté
  - Nom du script
  - Auteur
  - Date
  - Description
  - Si utilisé dans une cron, le précisé et la date d'exécution
  - Le chemin vers les erreurs, fichiers de log



# Example

```
#!/bin/bash
#
# Script Name: mytest.sh
#
# Author: Name of creator
# Date : Date of creation
#
# Description: The following script reads in a text file called /path/to/file
#              and creates a new file called /path/to/newfile
#
# Run Information: This script is run automatically every Monday of every week at 20:00hrs from
#                  a crontab entry.
#
# Error Log: Any errors or output associated with the script can be found in /path/to/logfile
#
```



# Débogage

- Ajouté au shebang -xv (x : cmd & Arg, v : num ligne)

```
#!/bin/bash -xv
```

- Sinon lancé le script avec :
  - sh -xv <votre\_script>
  - Bash -xv <votre\_script>
- Votre script va se lancer ligne par ligne, en vous l'affichant dans votre terminal

# Les variables

Les variables

Concaténation

isolation

substitution

# Les variables

- Une variable peut être représenté comme une boîte au lettre
- Elle a une adresse (le pointeur vers la ram)
- Elle a un nom (pour la rendre unique)
- Et elle a un contenu qui varie



# Les variables

- Pour définir une variable dans un script :
  - `foo="bonjour"`
- Pour appeler une variable on l'appelle précédé du signe « \$ »
  - Ex :
    - `echo $foo`
    - Stdout nous retourneras : bonjour

# Les variables

- Dans vos scripts, vous avez accès aux variables d'environnement :

- Ex :

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
echo $PATH
```

- Autre exemple :

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
/home/marchal
marchal
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
echo $PWD
echo $LOGNAME
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

- Ps : printenv

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ printenv
SHELL=/bin/bash
SESSION_MANAGER=local/marchal-OMEN-Laptop-15-en1xxx
QT_ACCESSIBILITY=1
COLORTERM=truecolor
```

# Concaténation

- La concaténation est l'opération de joindre 2 chaînes de caractères entre elles :
  - Ex :
    - foo="bonjour"
    - Faa="le monde !"
    - echo "\$foo\$faa"

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
Bonjour le monde !
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
foo="Bonjour"
faa="le monde !"
echo "$foo $faa"
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ==
```



# Isolation

- L'isolation c'est le fait de différencier expréssement vos variables

– Ex :

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
Mes chien sont vieux
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
animale="chien"
echo "Mes $animale sont vieux"
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

Pourquoi, voulons nous isoler nos variables ?

Par exemple: J'aimerais pouvoir rendre chien au pluriel.  
Et retourner « Mes chiens sont vieux »

# Isolation

- Ex1 : 

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
Mes  sont vieux
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
animale="chien"
echo "Mes $animales sont vieux"
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

- Ex2 : 

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
Mes chien s sont vieux
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
animale="chien"
echo "Mes $animale s sont vieux"
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```



# Isolation

- Mais en entourant ma variable d'accolade, cela prend enfin le comportement attendu :

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
Mes chiens sont vieux
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
animale="chien"
echo "Mes ${animale}s sont vieux"
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

- C'est l'isolation !

# Substitution

- La substitution c'est travailler directement sur la valeur de votre variable pour modifier son rendu, sans modifier sa valeur.
  - `foo=coucou`
  - `echo "${#foo}"` # retourne le nombre de caractère : 6  
# pourtant foo vaut toujours «coucou»

# Substitution

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
chemin/vers/hello.txt
hello.txt
22
HELLO WORLD!
hello world!
Hello world!
HELLO WORLD!
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
STR="/chemin/vers/hello.txt"

echo ${STR#*/}      # chemin/vers/hello.txt
echo ${STR##*/}     # hello.txt
echo ${#STR}        # La longueur de STR

STR="HELLO WORLD!"
echo ${STR,}        #=> "hello WORLD!" (lowercase 1st letter)
echo ${STR,,}       #=> "hello world!" (all lowercase)

STR="hello world!"
echo ${STR^}        #=> "Hello world!" (uppercase 1st letter)
echo ${STR^^}       #=> "HELLO WORLD!" (all uppercase)
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

# Intérargir avec un script

read

le passage d'argument

set

shift

# read

- read avec un seule Arg

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ read ma_variable  
je suis une variable écrite à la suite de cette commande  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $ma_variable  
je suis une variable écrite à la suite de cette commande  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ read ma_variable  
je permet d'être interactif !  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $ma_variable  
je permet d'être interactif !  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ read ma_variable  
avec un script  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $ma_variable  
avec un script  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```



# read

- read avec plusieurs Arg

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ read ma_premiere_variable ma_seconde_variable
C'est aussi simple ?
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $ma_premiere_variable
C'est
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $ma_seconde_variable
aussi simple 1 z
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo "$ma_seconde_variable"
aussi simple ?
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ █
```

- L'espace fait office de délimiteur dans la réponse.

# Le passage d'argument

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
#!/bin/bash

echo $1
echo $2
echo $3
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh

marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut
Salut

marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A
Salut
A

marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous
Salut
A
Tous
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous Sa
Salut
A
Tous
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

# Le passage d'argument

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
```

```
#!/bin/bash
```

```
echo $@ #symbolise l'ensemble des arguments
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut
```

```
Salut
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A
```

```
Salut A
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous
```

```
Salut A Tous
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va ?
```

```
Salut A Tous, sa va 1 z
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va "?"
```

```
Salut A Tous, sa va 1 z
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va \?
```

```
Salut A Tous, sa va 1 z
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
```

```
#!/bin/bash
```

```
echo "$@" #symbolise l'ensemble des arguments
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va \?
```

```
Salut A Tous, sa va ?
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```



# set

- La commande set va modifier le comportement par défaut de vos scripts, via ce que l'on appelle :
  - Des drapeaux («Flags » en anglais)
    - Vous en avez déjà vu 2, le -x et -v ou -xv pour déboguer.
    - -C (majuscule) : empêche les redirections « > »(stdout) d'écraser vos fichiers

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
#!/bin/bash
set -C

echo "$@" > reponse_du_echo # "@" symbolise l'ensemble des arguments
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va \?
./test.sh: ligne 4: reponse_du_echo : impossible d'écraser le fichier existant
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

# set

- Pour enlever un drapeau : +

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
#!/bin/bash
set +C

echo "$@" > reponse_du_echo # "@" symbolise l'ensemble des arguments
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va ?
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat reponse_du_echo
Salut A Tous, sa va 1 z
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh Salut A Tous, sa va \?
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat reponse_du_echo
Salut A Tous, sa va ?
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

# shift

- La commande shift est a utilisé si vous ne voulez pas mettre de limite d'arg

```
#!/bin/bash

# This script can clean up files that were last accessed over 365 days ago.

USAGE="Usage: $0 dir1 dir2 dir3 ... dirN"

if [ "$#" == "0" ]; then
    echo "$USAGE"
    exit 1
fi

while (( "$#" )); do
    if [[ $(ls "$1") == "" ]]; then
        echo "Empty directory, nothing to be done."
    else
        find "$1" -type f -a -atime +365 -exec rm -i {} \;
    fi
    shift
done
```

# Les tests, les opérateurs if et case

Le code de retour \$?

Les opérateurs && et ||

La commande test.

L'utilisation des tests conditionnels if et case.

# Le code de retour \$ ?

- Code de retour / code de sortie (return code/exit code)
  - echo \$?
    - Renvoie toujours le dernier code de retour
      - 0 = succès
      - 1 - 255 = echec

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat reponse_du_echo
```

```
Salut A Tous, sa va ?
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $?
```

```
0
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ech
```

La commande « ech » n'a pas été trouvée, voulez-vous dire :

```
commande « echo » du deb coreutils (8.30-3ubuntu2)
```

```
commande « sch » du deb scheme2c (2012.10.14-1ubuntu1)
```

```
commande « ecs » du deb ecere-dev (0.44.15-1build3)
```

```
commande « bch » du deb bikeshed (1.78-0ubuntu1)
```

```
commande « ecl » du deb ecl (16.1.3+ds-4)
```

```
commande « dch » du deb devscripts (2.20.2ubuntu2)
```

```
commande « ecp » du deb ecere-dev (0.44.15-1build3)
```

```
commande « ecj » du deb ecj (3.16.0-1)
```

```
commande « ecm » du deb gmp-ecm (7.0.4+ds-5)
```

```
commande « ecc » du deb ecere-dev (0.44.15-1build3)
```

Essayez : `sudo apt install <nom du deb>`

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $?
```

```
127
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ tr
```

```
tr: opérande manquant
```

```
Saisissez « tr --help » pour plus d'informations.
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $?
```

```
1
```

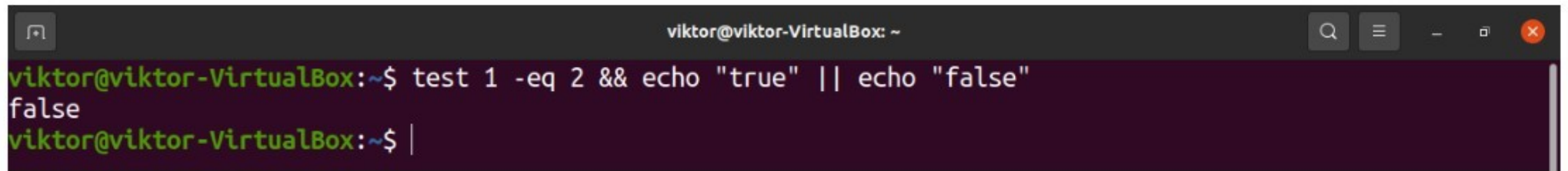
```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```



# test

- La commande test prend une expression en argument, la calcule et retourne son résultat

– Ex : `$ test 1 -eq 2 && echo "true" || echo "false"`

A screenshot of a terminal window titled 'viktor@viktor-VirtualBox: ~'. The prompt is 'viktor@viktor-VirtualBox:~\$'. The command entered is 'test 1 -eq 2 && echo "true" || echo "false"'. The output is 'false'. The prompt is now 'viktor@viktor-VirtualBox:~\$ |'.

```
viktor@viktor-VirtualBox:~$ test 1 -eq 2 && echo "true" || echo "false"
false
viktor@viktor-VirtualBox:~$ |
```

[Si 1 est égale à 2] alors affiche Vrai sinon affiche Faux

- Arg1 : le premier élément à comparer
- Arg2 : la méthode de comparaison
- Arg3 : le second élément à comparer

# Les opérateurs && et ||

- Ces 2 symboles spéciaux (métacharactère) représente la logique du :
  - && «et» Le temps est gris ET il pleut
    - Les 2 ne sont pas incompatibles
  - || « ou » Il pleut OU il ne pleut pas, j'ai bien dormi OU mal dormi
    - Les 2 SONT incompatibles



# test

- La commande test accepte ces éléments de comparaison :
  - -n <chaîne\_de\_caractère> # la longueur de \$ n'est pas 0
  - <chaîne\_de\_caractère> # la longueur de \$ n'est pas 0
  - -z <chaîne\_de\_caractère> # la longueur de \$ est 0
  - <une\_chaîne\_de\_caractère> = <autre\_chaîne\_de\_caractère> # les 2 chaînes se valent ex (ab = ab)=true
  - <une\_chaîne\_de\_caractère> != <autre\_chaîne\_de\_caractère> # les 2 chaînes NE se valent PAS ex (ab = ab)=true

# test

- Vous pouvez également utiliser test sans préciser le retour à faire.

Syntax of the test command	Questions asked
<code>test -d <i>pathname</i></code>	Is <i>pathname</i> a directory?
<code>test -f <i>pathname</i></code>	Is <i>pathname</i> a file?
<code>test -r <i>pathname</i></code>	Is <i>pathname</i> readable?
<code>test -w <i>pathname</i></code>	Is <i>pathname</i> writable?

# test

Syntax of the test command	Questions asked
<code>test <i>file1</i> -ot <i>file2</i></code>	Is <i>file1</i> older than <i>file2</i> ?
<code>test <i>file1</i> -nt <i>file2</i></code>	Is <i>file1</i> newer than <i>file2</i> ?

Table 3. Using the test command to compare the values of two numbers

Syntax of the test command	Questions asked
<code>test <i>A</i> -eq <i>B</i></code>	Is <i>A</i> equal to <i>B</i> ?
<code>test <i>A</i> -ne <i>B</i></code>	Is <i>A</i> not equal to <i>B</i> ?
<code>test <i>A</i> -gt <i>B</i></code>	Is <i>A</i> greater than <i>B</i> ?
<code>test <i>A</i> -lt <i>B</i></code>	Is <i>A</i> less than <i>B</i> ?
<code>test <i>A</i> -ge <i>B</i></code>	Is <i>A</i> greater than or equal to <i>B</i> ?
<code>test <i>A</i> -le <i>B</i></code>	Is <i>A</i> less than or equal to <i>B</i> ?

# test

- La commande test par default n'envoie pas de retour.

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ test 4 -lt 6
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $?
0
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ test 4 -gt 6
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $?
1
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

Il faut venir regarder le dernier message de retour,  
via echo \$ ?

# Condition if

- Les conditions, vont vous permettre d'exécuter du code en fonction de quelque chose.

```
if_example.sh
1.  #!/bin/bash
2.  # Basic if statement
3.
4.  if [ $1 -gt 100 ]
5.  then
6.      echo Hey that\'s a large number.
7.      pwd
8.  fi
9.
10. date
```

# Condition if `[[ ]]`

- `[ ]`, raccourci pour la commande `test`

`[[ ]]`, est une amélioration récente :

- vous fournit des opérateurs de comparaison en +
  - `=~` : pour match une regex
- vous permet d'utiliser `&&` et `||` de façon plus poussé



# Case

- Case va vous permettre
- D'énumérer simplement
- les possibilités.
- Si a vaut valeur, alors:
  - Oui
  - oui
  - O
  - Non
  - non
  - n

```
case $variable-name in
    pattern1)
        command1
        ...
        ....
        commandN
        ;;
    pattern2)
        command1
        ...
        ....
        commandN
        ;;
    patternN)
        command1
        ...
        ....
        commandN
        ;;
*)
esac
```

# Les boucles

La boucles for

While

Until

Les insructions

- Break
- Continue
- Exit

# while

- La boucle while, vont vous permettre de répéter une même action tant que sa condition vaut « True »

```
#!/bin/bash
x=1
while [ $x -le 5 ]
do
    echo "Welcome $x times"
    x=$(( $x + 1 ))
done
```

⚠ ⚠ Une boucle peut être infinie ⚠ ⚠

# until

- Ressemble a « while », until va se répéter tant que sa condition est « False »

```
#!/bin/bash

counter=0

until [ $counter -gt 5 ]
do
    echo Counter: $counter
    ((counter++))
done
```

⚠ ⚠ Une boucle peut être infinie ⚠ ⚠

# for

- For it  ere pour N fois d  finie par la condition, for incr  mente N automatiquement    chaque it  ration
  - sur un entier
  - Une liste
  - Une liste retourn  e par une cmd

```
1  for VARIABLE in 1 2 3 4 5 .. N
2  do
3      command1
4      command2
5      commandN
6  done
```

OR

```
1  for VARIABLE in file1 file2 file3
2  do
3      command1 on $VARIABLE
4      command2
5      commandN
6  done
```

OR

```
1  for OUTPUT in $(Linux-Or-Unix-Command-Here)
2  do
3      command1 on $OUTPUT
4      command2 on $OUTPUT
5      commandN
6  done
```



# Instructions break, continue, exit

- `break [n]`: termine la boucle en cours
  - Par default `n` vaut `1`, si vous avez une boucle dans une autre boucle `n=2` termine également la boucle parente, si `n=3` cloture la boucle dans la boucle dans la boucle etc...
- `continue [n]` : default = `1`, passe directement à la prochaine itération, lorsque `[n]` est donné, la nième boucle est reprise.
- `exit [code]`: arrête le programme et retourne le code définie, ex : `exit 0 ; echo $? ; # retourne 0`

# Le traitement arithmétique

Instruction expr

Instruction let

Instruction bc

# L'instructions expr

- La commande `expr` va vous permettre de faire des calculs sur vos variables
  - Syntaxe
    - `expr var1 + var2` = (l'addition de `var1 + var2`)
    - `expr var1 - var2` = (la soustraction de `var1 - var2`)
    - `expr var1 / var2` = ( la division de `var1` par `var2`)
    - `expr var1 \* var2` = (la multiplication de `var1` par `var2`)
      - On échape le signe multiplication avec le caractère d'échappement « `\` », car pour linux il symbolise aussi « tout ».

# Instructions let

- let évalue chaque argument, arg, comme une expression mathématique. Les arguments sont évalués de gauche à droite.

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a=5
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a+=1
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $a
6
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a-=1
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $a
5
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a++
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $a
6
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a--
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $a
5
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a + 13
bash: let: + : erreur de syntaxe : opérande attendu (le symbole erroné est « + »)
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a+13
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $a
5
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ let a=a+13
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo $a
18
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

# Instructions bc

- bc pour (basic calculator) est une commande utile pour calculer à la volée

```
Input : $ echo "12+5" | bc
```

```
Output : 17
```

```
Input : $ echo "10^2" | bc
```

```
Output : 100
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./ad.sh
2

marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat ad.sh
#!/bin/bash

echo "12/5" | bc
echo $x
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```



# Le traitement des chaînes de caractères

Typage d'une variable

Manipulation des chaînes de caractères.



# Le traitement des chaines de caractères

- Bash supporte des opérations de manipulation sur les chaînes de caractères.
- Certains viennent de la substitution.
- les autres font partie des fonctionnalités de la commande UNIX expr.
- Sans parler de différents autres programmes

Ceci produit une syntaxe de commande non unifiée et des fonctionnalités qui se recoupent, sans parler de la confusion engendrée.

# Manipuler les chaines de caractères

- `blabla=abcABC123ABCabc`

- `echo ${blabla:0}`

`# abcABC123ABCabc`

- `echo ${blabla:1}`

`# bcABC123ABCabc`

- `echo ${blabla:7}`

`# 23ABCabc`

- `echo ${blabla:7:3}`

`# 23A, récupère les 3`  
`# premier caractère à partir`  
`# du 7 ème caractère`

# Typage d'une variable

- **-r lecture seule**

```
declare -r var1
```

. Une tentative de modification de la valeur d'une variable en lecture seule échoue avec un message d'erreur.

- **-i entier**

```
declare -i nombre
# Ce script va traiter les occurrences suivantes de "nombre" comme un entier.

nombre=3
echo "Nombre = $nombre"      # Nombre = 3

nombre=trois
echo "Nombre = $nombre"      # Nombre = 0
# Essaie d'évaluer la chaîne "trois" comme un entier.
```

# Les fonctions

La déclaration

Le passage d'arguments

Le mot clé return

L'externalisation des fonctions.

# Déclarer une fonction

- Une fonction est un bloc logique, réutilisable.
- Pour déclarer une fonction :

```
nom_de_la_fonction() {  
    commandes  
}
```

En une seule ligne :

```
nom_de_la_fonction() {commandes; }
```

# Passage d'argument dans une fonction

- De la même façon qu'à une commande

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh
salut a tous !
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
#!/bin/bash

affiche_arg_donné(){
echo "$@"
}

affiche_arg_donné salut a tous !
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh nononono
salut a tous !
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```



# Le mot clé return

- return, termine l'instruction en cours et renvoie la valeur défini

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test.sh
#!/bin/bash

affiche_arg_donné(){

return 12
echo "$@"

}

affiche_arg_donné salut a tous !
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ ./test.sh nononono
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ echo "$?"
12
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

# L'externalisation des fonctions

- Via source vous pouvez appelé des fonctions depuis d'autres fichiers
- `source <nom_de_fichier> [Argument]`
- Si `<nom_de_fichier>` n'est pas un chemin d'accès complet à un fichier
  - la commande recherchera dans `$PATH`.
  - Si le fichier n'est pas trouvé dans `$PATH`
    - Source ira chercher dans le répertoire actuel.

# L'externalisation des fonctions

- Si des arguments sont passées, ils deviendront les arguments du fichier
- Si le fichier existe retourne le code 1, sinon 0

```
#!/bin/bash
# -----
# [Author]                Thibault Marchal
# [Description]           Template de base pour script bash
# -----

Funct_a_appeler () {
echo "${@}"
}

main (){
Funct_a_appeler "${@}"
}

main "${@}"
marchal@marchal-OMEN-Laptop-15-en1xxx:~/script$ cat
demande_la_fonction.sh  funct_a_appeler.sh
marchal@marchal-OMEN-Laptop-15-en1xxx:~/script$ cat demande_la_fonction.sh
#!/bin/bash
# -----
# [Author]                Thibault Marchal
# [Description]           Template de base pour script bash
# -----

Demande_la_fonction () {
source funct_a_appeler.sh ${@}
}

main (){
Demande_la_fonction "${@}"
}

main "${@}"
marchal@marchal-OMEN-Laptop-15-en1xxx:~/script$ ./demande_la_fonction.sh
marchal@marchal-OMEN-Laptop-15-en1xxx:~/script$ ./demande_la_fonction.sh Salut a tous !
Salut a tous !
marchal@marchal-OMEN-Laptop-15-en1xxx:~/script$ █
```

# Expréssions régulières et commande grep

Exp régulière

grep

fgrep

egrep



# Exp régulière

- Rappel : une expression régulière décrit un ensemble de chaînes de caractères possibles.
- Les expressions régulières sont également appelées regex.



# catégorie

- Caractère :
  - \d : 1 chiffre de 0 à 9
  - \w : 1 lettre miniscule ou majuscule
  - \s : 1 un espace
  - \D : Qui n'est pas un chiffre, tout sauf un chiffre
  - \W : Qui n'est pas une lettre minuscule ou maj
  - \S : Qui n'est pas un espace
  - . : N'importe quel caractère sauf \n
  - \ : échape le caractère spécial ( \. = un point de fin de ligne)

# catégorie

- Quantificateurs (se cumule avec les autres attributs):
  - $+$  : 1 ou plus (du même attributs)
  - $*$  : 0 ou plus (du même ou d'autres attributs)
  - $\{3\}$  : Exactement 3 (du même attributs)
  - $\{2,4\}$  : Entre 2 et 4 fois (du même attributs)
  - $\{4,\}$  : 4 fois ou plus (du même attributs)
  - $?$  : 1 de plus (optionnel)

# Catégorie

- Logique :
  - | : Ou logique
  - ( ... ) : groupe de capture ex : P(omme|ort) = Pomme
  - \1 : groupe 1 ex : (\d\d)\+(\d\d)=\2\+\1
  - \2 : groupe 2
  - ( ?: ) : Dans un groupe, permet de rendre optionnel comparé au reste de l'expression
  - [a,b...] : Un seul caractère dans la liste
  - [v-w] : Un seul caractère entre v et w

# Catégorie

- [^a] : Qui n'est pas le caractère qui suit ^
- [^c-f] : Qui n'est pas entre a et z
- [\d\D] : Un caractère qui est ou n'est pas un chiffre

# Catégorie

- Ancre
- `^` : vérifie que c'est le debut de la ligne
  - `! !` quand entre crochet , verifie que le caractère suivant n'est pas le premier du mot/ligne `! !`
- `$` : Vérifie que le caractère suivant est la fin de ligne/ fin de mot
- `\A` : vérifie que le caractère suivant est le début d'un mot
- Etc..... (<https://www.rexegg.com/regex-quickstart.html#chars>)

# Grep, fgrep, egrep

- A l'origine des programmes distincts, maintenant majoritairement intégré à grep :

- Regarder la taille du binaire →

- 195 Ko pour grep
- 28 pour les 2 autres

```
marchal@marchal-OMEN-Laptop-15-  
grep  
195K grep  
28 egrep  
28 fgrep
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ strings /usr/bin/egrep  
#!/bin/sh  
exec grep -E "$@"  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat /usr/bin/egrep  
#!/bin/sh  
exec grep -E "$@"  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat /usr/bin/fgrep  
#!/bin/sh  
exec grep -F "$@"  
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

ou F:



# Grep

- Grep vous permet de trier stdout via mot clé

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep grep
grep, egrep, fgrep, rgrep - print lines that match patterns
grep [OPTION...] PATTERNS [FILE...]
grep [OPTION...] -e PATTERNS ... [FILE...]
grep [OPTION...] -f PATTERN_FILE ... [FILE...]
grep searches for PATTERNS in each FILE. PATTERNS is one or more patterns separated by
ERNS should be
quoted when grep is used in a shell command.
In addition, the variant programs egrep, fgrep and rgrep are the same as grep -E, grep -F,
for backward
Output the version number of grep and exit.
Interpret PATTERNS as Perl-compatible regular expressions (PCREs). This option is
unimplemented
Stop reading a file after NUM matching lines. If the input is standard in
ndard input is
positioned to just after the last matching line before exiting, regardless of the pr
. When grep
stops after NUM matching lines, it outputs any trailing context lines. When th
When the -v or
--invert-match option is also used, grep stops after outputting NUM non-matching lin
foo.gz | grep --label=foo -H 'some pattern'. See also the -H option.
Report Unix-style byte offsets. This switch causes grep to report byte offsets as
s will produce
```

# Grep options

- -i : Ignore la casse
- -v : Retourne tout ce qui ne contient pas le motif
- -w : Motif en début de ligne ou précédé d'un élément de ponctuation, pratique pour savoir ce que font vos options ;)

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep -w w
-w, --word-regexp
provided it's not at the edge of a word. The symbol \w is a s
```

- -x : Qui matche exactement une ligne entière

# Grep options

- Control de la sortie
  - `--color` : Colorie ce qui match
  - `-l` : Signale visuellement si le motif match

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep -l -- "et"
(entrée standard)
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep "et"
      Interpret PATTERNS as extended regular expressions (EREs,
      Interpret PATTERNS as fixed strings, not regular expressions.
      Interpret PATTERNS as basic regular expressions (BREs, see
      Interpret PATTERNS as Perl-compatible regular expressions.
```

- `-L` : Inverse de `l` signale si le motif de match pas
- `-m` : Arrete de chercher apres N occurrence, ex : `-m 2`

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep -m 2 -- "et"
      Interpret PATTERNS as extended regular expressions (EREs, see below).
      Interpret PATTERNS as fixed strings, not regular expressions.
```

# Grep options

- -q : Pour quiet ( --quiet marche aussi), return avec 0 status si un match est trouvé
- -n : Affiche les lignes

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep -n -m 2 -- "et"
30:      Interpret PATTERNS as extended regular expressions (EREs, see below).
33:      Interpret PATTERNS as fixed strings, not regular expressions.
```

- -c : Compte le nombre d'occurrence

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | fgrep -c -- "et"
64
```

# Fgrep, Egrep

- Fgrep ou grep -f :
  - Cherche une chaîne de caractère précise
- Egrep ou grep -e :
  - Chercher avec une regex :

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ man grep | grep FI
grep [OPTION...] PATTERNS [FILE...]
grep [OPTION...] -e PATTERNS ... [FILE...]
grep [OPTION...] -f PATTERN_FILE ... [FILE...]
grep searches for PATTERNS in each FILE. PATTERNS
```

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ egrep '^no(fork|group)' /etc/group
nogroup:x:65534:
```



# Catégorie

- Par exemple :
  - $a.*z$  : tout chaîne qui commence par a et se termine par z
  - $[a-z]\{1,5\}$  : tous les mots de 1 à 5 lettres minuscules
  - $\backslash d\{2\} \backslash d\{2\} \backslash d\{4\}$  : toutes les dates au format jj/mm/aaaa (un peu plus)
  - $[a-z0-9.-]^+@[a-z0-9.-]^+(\.[a-z]^+)^*$  : toutes les adresses e-mail (un peu moins)





# La commande sed

La syntaxe, son potentiel  
quelques cas

# sed

- La commande sed est un editeur de stream/flux.
- Un équivalent tr en plus puissant

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test | grep linux
linux is great os. unix is opensource. unix is free os.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ sed 's/unix/linux/' test|grep linux
linux is great os. linux is opensource. unix is free os.
linux linux which one you choose.
linux is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

- Dans cette exemple on remplace « unix » par « linux »,  
1 fois par ligne

# sed

```
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ cat test | grep linux
linux is great os. unix is opensource. unix is free os.
linux linux which one you choose.
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
marchal@marchal-OMEN-Laptop-15-en1xxx:~$ sed 's/unix/linux/' test|grep linux
linux is great os. linux is opensource. unix is free os.
linux linux which one you choose.
linux is easy to learn.linux is a multiuser os.Learn unix .unix is a powerful.
marchal@marchal-OMEN-Laptop-15-en1xxx:~$
```

s/\_\_\_1\_\_\_/\_\_\_2\_\_\_/n, pour remplacer uniquement l'occurrence(n) du pattern

s/\_\_\_1\_\_\_/\_\_\_2\_\_\_/g, pour remplacer toutes les occurrences

# sed

- Si vous sentez que tr ne suffira pour votre besoin, alors sed fera l'affaire

# La commande awk

La syntaxe

les variables internes, les opérations.

# awk

- Awk est un langage pour la manipulation de texte
  - Syntaxe :
    - `awk [options] 'selection_criteria {action}'`
  - Opérations :
    - Analyser un fichier ligne par ligne.
    - Diviser la ligne/fichier d'entrée en champs.
    - Comparer la ligne ou les champs d'entrée avec le(s) modèle(s) spécifié(s).
    - Effectuer diverses actions sur les lignes correspondantes.
    - Formater les lignes de sortie.
    - Effectuer des opérations arithmétiques et de chaîne de caractères.
    - Utiliser le flux de contrôle et les boucles sur la sortie.
    - Transformer les fichiers et les données selon une structure spécifiée.
    - Générer des rapports formatés.