scriting

Astier Guillaume

06/01/2026

Fichier principal et fonctions

Math

Les flux

ReGex

Script

Qu'est-ce qu'un script

Au lieu de lancer les commandes directement dans un terminal, on peut écrire un fichier texte avec le shebang et les droits d'exécution

```
username@hostname:~$ cat myfirstscript.sh
#!/bin/bash

echo toto
username@hostname:~$ chmod +x myfirstscript.sh
username@hostname:~$ ./myfirstscript.sh
toto
```

Avantages désavantages

- **Avantages**

Avantages désavantages

- **Avantages**
- Plus lisible

Avantages désavantages

- **Avantages**
- Plus lisible
- Enregistré

Avantages désavantages

- **Avantages**
- Plus lisible
- Enregistré
- Exportable

Avantages désavantages

- **Avantages**
- Plus lisible
- Enregistré
- Exportable
- Débogage

Avantages désavantages

- **Avantages**
- Plus lisible
- Enregistré
- Exportable
- Débogage
- **Désavantages**

Avantages désavantages

- **Avantages**
- Plus lisible
- Enregistré
- Exportable
- Débogage
- **Désavantages**
- Débogage

Variable d'un script

Nom	Description
$0	le nom du programme shell courant.
$1...\{n\}$	les n paramètres passés au programme (au shell) lors de son appel.
$\#	le nombre de paramètres passés à l'appel du programme shell (sans inclure le paramètre $0)\|\|$*

Exemple d'utilisation

```
username@hostname:~$ cat mysecondscript.sh
#!/bin/bash
echo "Thx to launch ${0}"
echo "There are ${#} arguments"
echo "They are : ${*} but the second is $2"
false
echo ${?}

username@hostname:~$ ./mysecondscript.sh toto titi tutu
Thx to launch ./mysecondscript.sh
There are 3 arguments
They are : toto titi tutu but the second is titi
1
```

Kit de survie

Règles d'or

- Indentez votre script

Règles d'or

- Indentez votre script
- Commentez votre script

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage
- Déclarez votre variable au début de votre script

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage
- Déclarez votre variable au début de votre script
- Toujours tester vos entrées

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage
- Déclarez votre variable au début de votre script
- Toujours tester vos entrées
- Donnez à votre script un peu "d'air frais"

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage
- Déclarez votre variable au début de votre script
- Toujours tester vos entrées
- Donnez à votre script un peu "d'air frais"
- Testez la valeur de retour de vos commandes SHELL ($?)

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage
- Déclarez votre variable au début de votre script
- Toujours tester vos entrées
- Donnez à votre script un peu "d'air frais"
- Testez la valeur de retour de vos commandes SHELL ($?)
- Utilisez le man , niveau 1 (essayez un man -k)

Règles d'or

- Indentez votre script
- Commentez votre script
- Utilisez une règle de nommage
- Déclarez votre variable au début de votre script
- Toujours tester vos entrées
- Donnez à votre script un peu "d'air frais"
- Testez la valeur de retour de vos commandes SHELL ($?)
- Utilisez le man , niveau 1 (essayez un man -k)
- Rendez votre script exécutable : `chmod +x Monnouveauscript.sh`

Exemple de règles d'or 1/2

```
#BAD
if [[ -f $titi ]];then echo "your parameter is a file";cp $1 "$1".old;fi
#GOOD
if [[ -f ${Nom_Fichier_Saisi} ]]
then
echo "your parameter is a file"
cp $1 "${Nom_Fichier_Saisi}".old
fi
```

Exemple de règles d'or 2/2

si votre script attend un argument représentant un nom de fichier.

```
#Test of arguments
if [ $# -lt 1 ]
then
echo "You must give an argument for the script"
exit 1
fi
#Test of the type of the first argument
if [ -e $1 ]
then
echo "You must give an valid file name for the first argument for the script"
exit 2
fi
```

Algo

SI Condition

Condition IF

THEN

——> *Launch_action*

END IF

Exemple de condition if

```
username@hostname:~$ cat exampleIf.sh
#!/bin/bash
if [ $1 -eq 1 ]
then
echo "The first argument is 1"
fi
username@hostname:~$ bash exampleIf.sh 2

username@hostname:~$ bash exampleIf.sh 1
The first argument is 1
```

condition si/sinon

Condition IF

THEN

——> *Launch_action*

ELSE

——> *Launch_action*

END IF

Exemple de condition if/else

```
username@hostname:~$ cat exampleIfElse.sh
#!/bin/bash
if [ $1 -eq 1 ]
then
echo "The first argument is 1"
else
echo "The first argument is not 1"
fi
username@hostname:~$ exempleIfElse.sh 2
The first argument is not 1
username@hostname:~$ exempleIfElse.sh 1
The first argument is 1
```

condition if/elif

Condition IF

THEN

——> *Launch_action*

ELSE IF other_condition

THEN

——> *Launch_action*

END IF

Exemple de condition if/elif

```
username@hostname:~$ exampleIfelIf.sh
#!/bin/bash
if [ $1 -eq 1 ]
then
echo "The first argument is 1"
elif [ $1 -eq 2 ]
then
echo "The first argument is 2"
fi
username@hostname:~$ exampleIfelIf.sh 10
username@hostname:~$ exampleIfelIf.sh 1
The first argument is 1
username@hostname:~$ exampleIfelIf.sh 2
The first argument is 2
```

condition if/elif/else

Condition IF

THEN

——> *Launch_action*

ELSE IF other_condition

THEN

——> *Launch_action*

ELSE

——> *Launch_action*

END IF

Exemple de condition if/elif/else

```
username@hostname:~$ exampleIfelIfElse.sh
#!/bin/bash
if [ $1 -eq 1 ]
then
echo "The first argument is 1"
elif [ $1 -eq 2 ]
then
echo "The first argument is 2"
else
echo "I do not understant"
fi
username@hostname:~$ exampleIfelIfElse.sh 10
I do not understant
username@hostname:~$ exampleIfelIfElse.sh 1
The first argument is 1
username@hostname:~$ exampleIfelIfElse.sh 2
The first argument is 2
```

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux
- "$entier1" -ne "$entier2" –> Les 2 nombres sont différents

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux
- "$entier1" -ne "$entier2" –> Les 2 nombres sont différents
- "$entier1" -le "$entier2" –> entier1 <= entier2

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux
- "$entier1" -ne "$entier2" –> Les 2 nombres sont différents
- "$entier1" -le "$entier2" –> entier1 <= entier2
- "$entier1" -lt "$entier2" –> entier1 < entier2

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux
- "$entier1" -ne "$entier2" –> Les 2 nombres sont différents
- "$entier1" -le "$entier2" –> entier1 <= entier2
- "$entier1" -lt "$entier2" –> entier1 < entier2
- "$entier1" -ge "$entier2" –> entier1 >= entier2

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux
- "$entier1" -ne "$entier2" –> Les 2 nombres sont différents
- "$entier1" -le "$entier2" –> entier1 <= entier2
- "$entier1" -lt "$entier2" –> entier1 < entier2
- "$entier1" -ge "$entier2" –> entier1 >= entier2
- "$entier1" -gt "$entier2" –> entier1 > entier2

Tests

Pour connaître tous les tests possibles `man test`

Les plus courantes :

- "$chaine1" = "$chaine2" –> Les 2 chaînes sont identiques
- "$chaine1" != "$chaine2" –> Les 2 chaînes ne sont pas identiques
- -z "$chaine" –> La chaîne est vide
- -n "$chaine" –> La chaîne n'est pas vide
- "$entier1" -eq "$entier2" –> Les 2 nombres sont égaux
- "$entier1" -ne "$entier2" –> Les 2 nombres sont différents
- "$entier1" -le "$entier2" –> entier1 <= entier2
- "$entier1" -lt "$entier2" –> entier1 < entier2
- "$entier1" -ge "$entier2" –> entier1 >= entier2
- "$entier1" -gt "$entier2" –> entier1 > entier2
- -e "$file" –> Le fichier existe

Règles de comparaison 1/3

Opérandes de fichier

Opérande	Description	exemple
-e nomfichier	true si le nom de fichier existe	[-e /etc/shadow]
-d nomfichier	true si le nom du fichier est un répertoire	[-d /tmp/trash]
-f nomfichier	true si le nom du fichier est un fichier ordinaire	[-f /tmp/Log.txt]
-L nomfichier	true si le nom du fichier est un lien symbolique	[-L /home]
-r nomfichier	true si le nom du fichier est lisible (r)	[-r /boot/vmlinuz]
-w nomfichier	true si le nom du fichier est modifiable (w)	[-w /var/log]
-x nomfichier	true si le nom du fichier est un exécutable (x)	[-x /sbin/halt]

Règles de comparaison 2/3

Opérandes de chaîne

Opérande	Description	exemple
-z chaine	vrai si la chaîne est vide	`[ -z "${VAR}"]`
-n chaine	vrai si la chaîne n'est PAS vide	`[ -n "${VAR}"]`
chaine1 ==	vrai si les deux chaînes sont	`[ "${VAR}"== "toto"]`
chaine2	égales	
chaine1 =	vrai si les deux chaînes sont	`[ "${VAR}"= "toto"]`
chaine2	égales	
chaine1 !=	vrai si les deux chaînes ne sont	`[ "${VAR}"!= "toto"]`
chaine2	PAS égales	

Règles de comparaison 3/3

Opérandes numériques

Opérande	Description	exemple
num1 -eq num2	égalité	[$Nombre -eq 42]
num1 -ne num2	pas égal	[$Number -ne 42]
num1 -lt num2	inférieur à ($<$)	[$Nombre -lt 42]
num1 -le num2	inférieur ou égal ($<=$)	[$Nombre -le 42]
num1 -gt num2	supérieur à ($>$)	[$Number -gt 42]
num1 -ge num2	supérieur ou égal ($>=$)	[$Number -ge 42]

Exemple de test (1/2)

```
#!/bin/bash
# directory exists ? 1/2
test -d /home/user
rc=$?
if [ $rc -ne 0 ]
then
echo "The directory /home/user does not exist"
fi
```

Exemple de test (2/2)

```
#!/bin/bash
# directory exists ? 2/2
if [ -d "/home/user" ]
then
echo "the directory /home/user exists"
fi
# comparison of 2 strings
if [ "toto" = "titi" ]
then
echo "toto is not equal to titi"
fi
```

Boucle tant que

Condition WHILE

DO

——> *Launch_action*

END DO

Exemple de boucle while (1/2)

```
username@hostname:~$ while.sh
#!/bin/bash
a=0
while [ $a -le 3 ]
do
echo '$a'
a=$(( $a + 1 ))
done

username@hostname:~$ while.sh
0
1
2
3
```

Exemple de boucle while (2/2)

```
while true
do
echo $RANDOM
done
```

Le bash est compilé en tant que monothread 64 bits. Avec cette commande, votre bash utilisera 100 % d'un cœur de processeur. Pour protéger votre CPU, mettez toujours une action “inutile/time-out”

```
while true
do
echo $RANDOM
sleep 1
done
```

pour la boucle

FOR variable IN value1 value2 value3

DO

——> *Launch_action*

END DO

Exemple de boucle for (1/2)

```
username@hostname:~$ for1.sh
#!/bin/bash
for var in 'value1' 'value2' 'value3'
do
echo "Var =  ${var}" ;
done

username@hostname:~$ for1.sh
Var = value1
Var = value2
Var = value3
```

Exemple de boucle for (2/2)

Pour se rapprocher du code C (cette syntaxe est peu utilisée en bash) :

```
username@hostname:~$ for2.sh
#!/bin/bash
for i in $(seq 0 2)
do
echo $i
done

username@hostname:~$ for2.sh
0
1
2
```

Cette syntaxe `$(seq 0 3)` est équivalente à `((i=0;i<=3;i++))`

Case/Esac

```
case ${vars} in
1)  command1
command1bis
;;
2)  command2
command2bis
;;
*)  commanddefault
commanddefault2
;;
esac
```

Exemple de Cas/Esac

```
username@hostname:~$ myScriptCase.sh
#!/bin/bash
case ${1} in
toto) echo "toto is a beautifull name";;
titi) echo "I prefer toto as a name";;
*) echo "i do not understand"
esac

username@hostname:~$myScriptCase.sh toto
toto is a beautifull name
username@hostname:~$myScriptCase.sh titi
I prefer toto as a name
username@hostname:~$myScriptCase.sh Loic
i do not understand
```

ARRÊTER/CONTINUER

```
username@hostname:~$ for3.sh
#!/bin/bash
for var in value1 value2 value3 value4 value5
do
[ "$var" = "value2" ] && continue
[ "$var" = "value4" ] && break
echo $var
done

username@hostname:~$ for3.sh
value1
value3
```

BREAK = arrêter la boucle

CONTINUE = passer à l'itération suivante

La gestion des processus

La gestion des processus

Linux étant un système multitâche, plusieurs programmes peuvent s'exécuter en même temps.

Lorsqu'un programme est démarré, un processus est créé. C'est une entité active qui possède des caractéristiques (priorité, registres, compteur ordinal, mémoire, etc.). Certaines caractéristiques peuvent changer avec le temps

Le système identifie les processus à l'aide d'un identifiant (PID u003d *P*process *ID*entification).

La gestion des processus sous Linux est dite hiérarchique.

Un processus peut lui-même créer un autre processus (fork + exec). Le processus créé est appelé un processus enfant. Le créateur est appelé le processus parent.

nice et renice

nice et renice

Les commandes nice et renice vous permettent de définir ou de modifier la priorité d'un processus. La plage de valeurs possibles va de -20 (priorité la plus favorable) à 19 (la moins favorable).

```
username@hostname:~$ nice -n -20 find / -type f -name "*.sh"
username@hostname:~$ renice 20 7643
```

kill

kill

La commande kill envoie un signal à un processus. Des “superpositions” à la commande kill existent killall, pgrep / pkill, xkill

```
username@hostname:~$ kill 456
username@hostname:~$ kill -9 -1
username@hostname:~$ pkill firefox
```

Gestion des tâches dans une session interactive

Gestion des tâches dans une session interactive

Les processus interactifs sont démarrés et gérés à partir du terminal de l'utilisateur. Il existe 2 modes :

- Mode premier plan

Gestion des tâches dans une session interactive

Les processus interactifs sont démarrés et gérés à partir du terminal de l'utilisateur. Il existe 2 modes :

- Mode premier plan
- Mode arrière-plan

Mode premier plan

Le processus monopolise le terminal jusqu'à sa terminaison

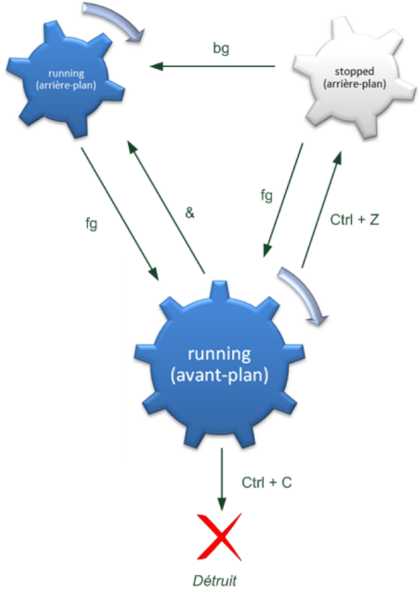
```
username@hostname:~$ sleep 10
[...]
```

Mode arrière-plan

Le processus fonctionne en parallèle avec le terminal

```
username@hostname:~$ sleep 10 &
[1] 3384
$
```

La séquence de touches “ctrl-z” et les commandes “jobs, bg, fg commands” permettent de faire basculer un processus d’un mode à l’autre.


running
(arrière-plan)
bg
stopped
(arrière-plan)
fg
&
fg
Ctrl + Z
running
(avant-plan)
Ctrl + C
Détruit

Afficher les processus

Afficher les processus

Vous pouvez utiliser la commande SHELL ps pour afficher tous les processus en cours d'exécution sur votre ordinateur

exemple pour voir les processus appartenant à votre ***SHELL*** actuel :

```
username@hostname:~$ ps
3837 pts/2    00:00:00 bash
137967 pts/2    00:00:09 evince
144605 pts/2    00:00:00 ps
```

exemple pour voir les processus vous appartenant ***actuel propriétaire*** :

```
username@hostname:~$ ps -u username
PID TTY          TIME CMD
2053 ?        00:00:02 systemd
2054 ?        00:00:00 (sd-pam)
2059 ?        00:04:16 pulseaudio
....
```

Comment avoir un code source élégant

Le code de retour

Il faut contrôler le code de retour ($?). Dans une fonction, il faut utiliser la commande `return <ReturnCode>`, et dans la *fonction principale* il faut utiliser la commande `exit <ReturnCode>`.

Attention, la commande `return` quitte immédiatement la fonction. La commande `exit` termine le script (même si elle est utilisée dans une fonction).

Vérifiez vos entrées

Quelle que soit la façon dont les données d'entrée sont récupérées, elles doivent toujours être vérifiées.

- si un fichier doit être lu, il faut déjà savoir s'il existe

Beaucoup de vérifications existent dans la commande `test` (voir les règles de comparaison).

Vérifiez vos entrées

Quelle que soit la façon dont les données d'entrée sont récupérées, elles doivent toujours être vérifiées.

- si un fichier doit être lu, il faut déjà savoir s'il existe
- si vous devez faire une opération arithmétique, il faut vérifier que ce sont bien des nombres (attention à la division par 0)

Beaucoup de vérifications existent dans la commande `test` (voir les règles de comparaison).

Vérifiez vos entrées

Quelle que soit la façon dont les données d'entrée sont récupérées, elles doivent toujours être vérifiées.

- si un fichier doit être lu, il faut déjà savoir s'il existe
- si vous devez faire une opération arithmétique, il faut vérifier que ce sont bien des nombres (attention à la division par 0)
- ...

Beaucoup de vérifications existent dans la commande `test` (voir les règles de comparaison).

Indentation

Préférez-vous ce code :

```
function action () { echo -ne "${1}\t";shift;${*};rc=${?};\
[ ${rc} -eq 0 ] && echo '[OK]' || \
echo '[KO]';return ${rc};}
```

ou celui-ci ?

```
function action () {
    echo -ne "${1}\t"
    shift
    ${*}
    rc=${?}
    [ ${rc} -eq 0 ] && echo '[OK]' || echo '[KO]'
    return ${rc}
}
```

Je préfère le second script. L'indentation est une question de goût. Mais elle doit au moins être homogène dans tout le script. Pensez à aérer votre code (une ligne vide ne coûte rien, mais rend le code plus lisible).

Commentaires

Ajoutez beaucoup de commentaires dans votre script. Pourquoi ?

- si vous reprenez votre code après 6 mois, vous le comprendrez plus facilement

Tous les projets en environnement de test / production sont basés sur des exigences.

Ces exigences sont nécessaires pour s'assurer que la création finale fait ce que nous/ils attendent.

Commentaires

Ajoutez beaucoup de commentaires dans votre script. Pourquoi ?

- si vous reprenez votre code après 6 mois, vous le comprendrez plus facilement
- si vous donnez votre code à quelqu'un d'autre, il le comprendra plus facilement

Tous les projets en environnement de test / production sont basés sur des exigences.

Ces exigences sont nécessaires pour s'assurer que la création finale fait ce que nous/ils attendent.

Commentaires

Ajoutez beaucoup de commentaires dans votre script. Pourquoi ?

- si vous reprenez votre code après 6 mois, vous le comprendrez plus facilement
- si vous donnez votre code à quelqu'un d'autre, il le comprendra plus facilement
- l'utilisateur de votre code comprendra plus facilement les erreurs

Tous les projets en environnement de test / production sont basés sur des exigences.

Ces exigences sont nécessaires pour s'assurer que la création finale fait ce que nous/ils attendent.

Commentaires

Ajoutez beaucoup de commentaires dans votre script. Pourquoi ?

- si vous reprenez votre code après 6 mois, vous le comprendrez plus facilement
- si vous donnez votre code à quelqu'un d'autre, il le comprendra plus facilement
- l'utilisateur de votre code comprendra plus facilement les erreurs
- ...# Construction de Script

Tous les projets en environnement de test / production sont basés sur des exigences.

Ces exigences sont nécessaires pour s'assurer que la création finale fait ce que nous/ils attendent.

Exigences

Exigences

Ce script a pour but de classer les anciens élèves présents lors d'une réunion.

Ici, nous allons créer un script avec ces exigences :

- Le nom du script est : alumni-reunion.sh

Exigences

Ce script a pour but de classer les anciens élèves présents lors d'une réunion.

Ici, nous allons créer un script avec ces exigences :

- Le nom du script est : alumni-reunion.sh
- Le script doit analyser un fichier csv passé en premier argument

Exigences

Ce script a pour but de classer les anciens élèves présents lors d'une réunion.

Ici, nous allons créer un script avec ces exigences :

- Le nom du script est : alumni-reunion.sh
- Le script doit analyser un fichier csv passé en premier argument
- Le format du fichier csv est : NOM;PRENOM;ANNEE;NIVEAU (chemin du fichier : /data/admin/list/student-list)

Exigences

Ce script a pour but de classer les anciens élèves présents lors d'une réunion.

Ici, nous allons créer un script avec ces exigences :

- Le nom du script est : alumni-reunion.sh
- Le script doit analyser un fichier csv passé en premier argument
- Le format du fichier csv est : NOM;PRENOM;ANNEE;NIVEAU (chemin du fichier : /data/admin/list/student-list)
- Pour chaque ligne du fichier csv, le script doit demander : "PRENOM NOM est présent ? (y/n)"

- La réponse doit être y ou n et il ne faut pas utiliser la touche Entrée pour valider

- La réponse doit être y ou n et il ne faut pas utiliser la touche Entrée pour valider
- Le script doit créer un dossier avec la date (ex : /data/admin/result/2022-09-27)

- La réponse doit être y ou n et il ne faut pas utiliser la touche Entrée pour valider
- Le script doit créer un dossier avec la date (ex : /data/admin/result/2022-09-27)
- À chaque réponse y ou n, il faut ajouter dans le premier argument un nouveau fichier (chemin : /data/admin/result/YYYY-MM-DD/)

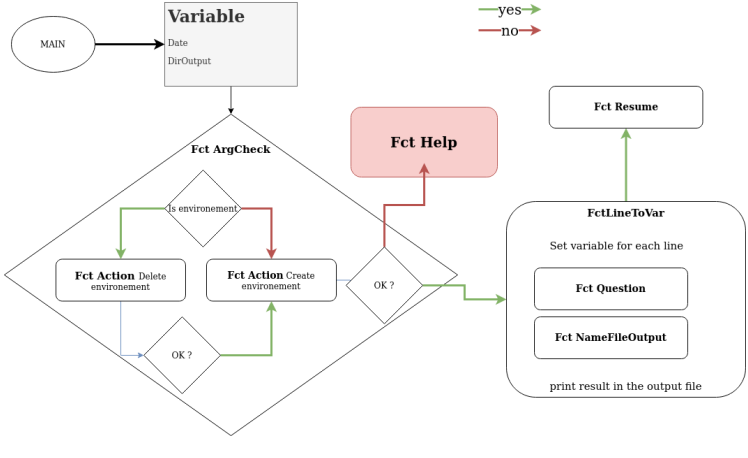
- La réponse doit être y ou n et il ne faut pas utiliser la touche Entrée pour valider
- Le script doit créer un dossier avec la date (ex : /data/admin/result/2022-09-27)
- À chaque réponse y ou n, il faut ajouter dans le premier argument un nouveau fichier (chemin : /data/admin/result/YYYY-MM-DD/)
- Les fichiers de résultats sont triés par groupe [année][niveau] (ex : /data/admin/result/YYYY-MM-DD/student-list_2020-M1)

- La réponse doit être y ou n et il ne faut pas utiliser la touche Entrée pour valider
- Le script doit créer un dossier avec la date (ex : /data/admin/result/2022-09-27)
- À chaque réponse y ou n, il faut ajouter dans le premier argument un nouveau fichier (chemin : /data/admin/result/YYYY-MM-DD/)
- Les fichiers de résultats sont triés par groupe [année][niveau] (ex : /data/admin/result/YYYY-MM-DD/student-list_2020-M1)
- Le format du contenu des fichiers est : NOM;PRENOM;ANNEE;NIVEAU;[present|absent]

- La réponse doit être y ou n et il ne faut pas utiliser la touche Entrée pour valider
- Le script doit créer un dossier avec la date (ex : /data/admin/result/2022-09-27)
- À chaque réponse y ou n, il faut ajouter dans le premier argument un nouveau fichier (chemin : /data/admin/result/YYYY-MM-DD/)
- Les fichiers de résultats sont triés par groupe [année][niveau] (ex : /data/admin/result/YYYY-MM-DD/student-list_2020-M1)
- Le format du contenu des fichiers est : NOM;PRENOM;ANNEE;NIVEAU;[present|absent]
- Le script se termine par un résumé de tous les fichiers créés et le total des présents/absents (avec un en-tête : FICHIER | présent | absent |)

Algorithme

Algorithme


MAIN
Variable
Date
DirOutput
yes
no
Fct ArgCheck
Is environement
Fct Action Delete environement
Fct Action Create environement
OK ?
OK ?
Fct Help
Fct Resume
FctLineToVar
Set variable for each line
Fct Question
Fct NameFileOutput
print result in the output file

Fonctions Classiques

Fonctions Classiques

Les fonctions classiques nécessaires dans le script sont les suivantes :

- Check ou Action pour vérifier chaque commande

Fonctions Classiques

Les fonctions classiques nécessaires dans le script sont les suivantes :

- Check ou Action pour vérifier chaque commande
- Help ou Usage pour afficher les informations sur le script

Fonctions Classiques

Les fonctions classiques nécessaires dans le script sont les suivantes :

- Check ou Action pour vérifier chaque commande
- Help ou Usage pour afficher les informations sur le script
- ArgCheck pour vérifier l'argument ou le type d'argument

Fonctions Spécifiques

Fonctions Spécifiques

Pour chaque script, il faut analyser les exigences pour les traduire en fonctions ou en code bash.

- LineToVar : traduit chaque colonne d'une ligne en une variable spécifique (Nom=[...] Prenom=[...])

Fonctions Spécifiques

Pour chaque script, il faut analyser les exigences pour les traduire en fonctions ou en code bash.

- LineToVar : traduit chaque colonne d'une ligne en une variable spécifique (Nom=[...] Prenom=[...])
- Question : pose à l'utilisateur la question "PRENOM NOM est présent ? (y/n)" et affiche le résultat sur la sortie standard de la fonction

Fonctions Spécifiques

Pour chaque script, il faut analyser les exigences pour les traduire en fonctions ou en code bash.

- LineToVar : traduit chaque colonne d'une ligne en une variable spécifique (Nom=[...] Prenom=[...])
- Question : pose à l'utilisateur la question "PRENOM NOM est présent ? (y/n)" et affiche le résultat sur la sortie standard de la fonction
- NameFileOutput : crée la variable du fichier de sortie /data/admin/result/student-list_ANNEE-NIVEAU avec les données de LineToVar et le résultat de Question

Fonctions Spécifiques

Pour chaque script, il faut analyser les exigences pour les traduire en fonctions ou en code bash.

- LineToVar : traduit chaque colonne d'une ligne en une variable spécifique (Nom=[...] Prenom=[...])
- Question : pose à l'utilisateur la question "PRENOM NOM est présent ? (y/n)" et affiche le résultat sur la sortie standard de la fonction
- NameFileOutput : crée la variable du fichier de sortie /data/admin/result/student-list_ANNEE-NIVEAU avec les données de LineToVar et le résultat de Question
- Resume : affiche tous les fichiers créés avec le total des présents/absents (ex : FICHIER : 12 2)

Sortie

Sortie (Premier lancement)

```
user@pem ~ $ ./alumni-reunion.sh list-student.csv

* Création de l'environnement /data/admin/result/2022-09-28 : OK
KADE Anthony est présent ? (y/n) : y
LILIAN Giles est présent ? (y/n) : y
[...]
ASIA Petty est présent ? (y/n) : n

FICHIER                 |       présent |       absent
------------------------------------------------------------
student-list_2013-M2    |       0       |       1
student-list_2018-M2    |       1       |       0
student-list_2020-M2    |       1       |       0
------------------------------------------------------------
TOTAL                   |       2       |       1
```

Sortie (relance)

```
user@pem ~ $ ./alumni-reunion.sh /data/admin/list/student-list
/data/admin/result/2022-09-28 existe \
voulez-vous continuer (supprimer toutes les données) (y/n) : y
* Nettoyage de l'environnement : OK
KADE Anthony est présent ? (y/n) : y
LILIAN Giles est présent ? (y/n) : y
[...]
ASIA Petty est présent ? (y/n) : n

FICHIER                 |       présent |       absent
------------------------------------------------------------
student-list_2013-M2    |       0       |       1
student-list_2018-M2    |       1       |       0
student-list_2020-M2    |       1       |       0
------------------------------------------------------------
TOTAL                   |       2       |       1
```

Fonction

Fonction

RAPPEL : TOUTES les fonctions doivent être en haut du script avant le main !!!

Help

La fonction Help sert uniquement à afficher les informations et à quitter. Si quelque chose ne va pas, vous pouvez utiliser cette fonction pour quitter le script.

```
# Print the help and exit
function Help() {
        echo "$(basename $0) [absolute or relative path file]"
        [[ ! -z $1 ]] && [[ $(let $1) ]] && Exit=$1 || Exit=0
        exit ${Exit}
}
```

Action

La fonction Action prend 2 arguments :

- ce qu'il faut afficher

Action

La fonction Action prend 2 arguments :

- ce qu'il faut afficher
- la commande à exécuter (pas de stdout/stderr)

```
# Print info exec and status
function Action () {
        # Print the first arguement without \n in the end (-n)
        echo -ne "\n\* $1 : "
        # Shift to the left
        shift
        # execute all the argument like a classic command but redirect in /dev/
            null
        $* &> /dev/null
        ResultExec=$?
        # Check the result and print OK/Failed
        if [[ ${ResultExec} -eq 0 ]]
                then
                        echo OK
                else
                        echo Failed; Help ${ResultExec}
        fi
}
```

Check

Vous pouvez utiliser Check ou Action mais Check s'utilise différemment.

```
function Check() {
        HaveToExit=$2
        [[ ${HaveToExit} -eq 1 ]] && $Help ${ResultExec}
        if [[ $1 -eq 0 ]]
                then
                        echo OK
                else
                        echo Failed
                        Help $1
        fi
}
# Exemple :
echo -n 'Is it OK ? : '
true
Check $? 0
```

ArgCheck

Vérifie l'environnement :

- le premier argument est-il un fichier

ArgCheck

Vérifie l'environnement :

- le premier argument est-il un fichier
- le dossier de sortie existe-t-il

ArgCheck

Vérifie l'environnement :

- le premier argument est-il un fichier
- le dossier de sortie existe-t-il
- la création du dossier de sortie est-elle OK

```
function ArgCheck() {
        # Check the first arg of the script and directory output data
        [[ ! -f $1 ]] && echo "$1 is not a file" && Help 1
        if [[ -d ${DirOutput} ]]; then
                while [[ $Qans != "y" ]] && [[ $Qans != "n" ]]
                        do
                                echo -e "\n"
                                read -p  "${DirOutput} exist \
do you want to continue (delete all data) (y/n) : " -n1 Qans
                        done
                # Clean environement
                [[ $Qans == "n" ]] && exit || \
                Action "Clean environement" "1" rm -rf ${DirOutput}/*
        fi
}
```

NameFileOutput

Crée la variable qui contient le nom du fichier de sortie pour chaque ligne du fichier d'entrée

```
# Create the varname of the output file
function NameFileOutput(){
        FileName=student-list_${1}-${2}
        echo ${DirOutput}/${FileName}
}
```

Question

Pour chaque ligne du fichier d'entrée, il faut poser la question présent/absent

```
function Question() {
        qName=$1
        qSurname=$2
        Answer=""
        Result=""
        # Check if the data is y or n
        while [[ -z ${Result} ]]
                do
                        read -p "$Name $Surname is present ? (y/n) : " \
                        -n1 Answer </dev/tty
                        [[ ${Answer} == "y" ]] && Result=present
                        [[ ${Answer} == "n" ]] && Result=absent
                done
        echo $Result
}
```

LineToVar

Analyse tout le fichier, récupère les données et appelle les autres fonctions

```
# Analyse for all line
function LineToVar(){
        while read -r Line
                do
                        echo -e "\n"
                        # Gen variable environement for each line
                        Name=$(echo $Line | cut -d";" -f1)
                        Surname=$(echo $Line | cut -d";" -f2)
                        Year=$(echo $Line | cut -d";" -f3)
                        Level=$(echo $Line | cut -d";" -f4)
                        Result=$(Question "${Name}" "${Surname}")
                        OutputFile=$(NameFileOutput "${Year}" "${Level}")
                        # output data in the outputfile
      echo "${Name};${Surname};${Year};${Level};${Result}" >> ${OutputFile}

                done  < $1
}
```

Resume

Affiche le résumé de tous les fichiers de sortie créés

```
function Resume() {
        echo -e "\n\nFILE\t\t\t|\tpresent\t|\tabsent"
        echo  "------------------------------------------------------------------"
        # find all output file
        for FileOutput in $(find ${DirOutput} -type f \
                                -name "student-list_*")
                do
                        # Gen variable for each file
                        FileName=$(basename $FileOutput)
                        Present=$(cat ${FileOutput} | grep present|wc -l)
                        Absent=$(cat ${FileOutput} | grep absent|wc -l)
                        TotPresent=$((TotPresent+Present))
                        TotAbsent=$((TotAbsent+Absent))
                        echo -e "$FileName\t|\t${Present}\t|\t$Absent"
                done
        echo  "------------------------------------------------------------------"
        # print total
        echo -e "TOTAL\t\t\t|\t${TotPresent}\t|\t$TotAbsent"
}
```

Main

Main appelle les fonctions et crée des variables pour tout le script et les fonctions

```
######## MAIN ########
Date=$(date "+%Y-%m-%d")

DirOutput=/data/admin/result/${Date}

ArgCheck $1

[[ ! -d ${DirOutput} ]] && \
Action "Create environement ${DirOutput}" "1" mkdir -p ${DirOutput}

LineToVar $1

Resume
```

Fichier principal et fonctions

Fichier principal et fonctions

Vous pouvez séparer le fichier principal et toutes les fonctions avec source

```
#!/bin/bash

source $(realpath $(dirname $0))/fct_classic
source $(realpath $(dirname $0))/fct_specific

ArgCheck $1

[[ ! -d ${DirOutput} ]] && \
Action "Create environement ${DirOutput}" "1" mkdir -p ${DirOutput}

LineToVar $1

Resume
```

Math

Maths dans le Bash

Les opérations arithmétiques en bash ne peuvent être effectuées que sur des nombres entiers.

```
$(( INT OPERATION_TYPE INT))

# Ex 1: simple addition
username@hostname:~$ echo $(( 2 + 2 ))
4
username@hostname:~$ foo=$(( 2 + 2 ))
username@hostname:~$ echo $foo
4
username@hostname:~$ echo $(( $foo + 2 ))
6
username@hostname:~$ echo $(( foo + 2 ))
6
```

Math types d'opérations dans Bash

Les types d'opérations sont :

- addition : '+'

Math types d'opérations dans Bash

Les types d'opérations sont :

- addition : '+'
- soustraction : -

Math types d'opérations dans Bash

Les types d'opérations sont :

- addition : '+'
- soustraction : -
- multiplication : *

Math types d'opérations dans Bash

Les types d'opérations sont :

- addition : '+'
- soustraction : -
- multiplication : *
- division : /

Arrondi en mathématiques bash

bash arrondit le résultat des opérations “vers le bas” pour obtenir un entier

```
username@hostname:~$ echo $(( 9 / 3 ))
3
username@hostname:~$ echo $(( 10 / 3 ))
3
username@hostname:~$ echo $(( 11 / 3 ))
3
username@hostname:~$ echo $(( 12 / 3 ))
4
```

bash / maths / bc

Si vous devez effectuer une opération arithmétique avec float, vous avez besoin de `bc` avec son option scale :

La syntaxe est la suivante :

```
username@hostname:~$ echo "scale=3;10/3" | bc
3.333
username@hostname:~$ echo "scale=2;10/3" | bc
3.33
username@hostname:~$ echo "scale=1;10/3" | bc
3.3
```

Séquences

Parfois, il est utile de générer une séquence d'entiers d'un début à un point final. Vous pouvez utiliser la commande `seq` pour le faire.

Exemple :

```
username@hostname:~$ seq 1 10
1
2
3
4
5
6
7
8
9
10
```

Les flux

Les flux

Chaque processus comporte 3 flux :

- le flux d'entrée *"stdin"*

Les flux

Chaque processus comporte 3 flux :

- le flux d'entrée *"stdin"*
- le flux de sortie standard *"stdout"*

Les flux

Chaque processus comporte 3 flux :

- le flux d'entrée *"stdin"*
- le flux de sortie standard *"stdout"*
- le flux de sortie d'erreur *"stderr"*

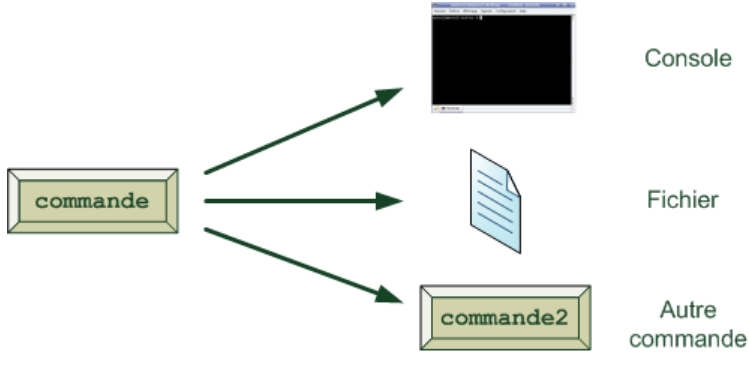


Figure 1: Flux Linux

sortie standard

C'est le flux le plus visible. Il sera affiche à l'écran.

Il est possible de :

- l'envoyer dans un nouveau fichier : `cmd > file`

sortie standard

C'est le flux le plus visible. Il sera affiche à l'écran.

Il est possible de :

- l'envoyer dans un nouveau fichier : `cmd > file`
- l'envoyer à la fin d'un fichier : `cmd >> file`

sortie standard

C'est le flux le plus visible. Il sera affiche à l'écran.

Il est possible de :

- l'envoyer dans un nouveau fichier : `cmd > file`
- l'envoyer à la fin d'un fichier : `cmd >> file`
- l'envoyer à une autre commande : `cmd1 |cmd2`

sortie standard

C'est le flux le plus visible. Il sera affiche à l'écran.

Il est possible de :

- l'envoyer dans un nouveau fichier : `cmd > file`
- l'envoyer à la fin d'un fichier : `cmd >> file`
- l'envoyer à une autre commande : `cmd1 |cmd2`
- Dans ce dernier cas, le stdout de *cmd1* sera le stdin de *cmd2*

commandes de chaînage

Avec ce dernier exemple vous pouvez “composer” une commande complexe en enchaînant plusieurs commandes simples. Exemple :

```
username@hostname:~$ echo -e "totontatantitintito" >>File.txt
username@hostname:~$ cat File.txt
toto
tata
titi
tito
username@hostname:~$ cat File.txt | grep "to"
toto
tito
username@hostname:~$ cat File.txt | grep "to" | wc -l
2
```

stderr

C'est le flux que le terminal affiche lors d'une erreur. Pour ce flux il est possible de :

- l'envoyer dans un nouveau fichier : `cmd 2> file`

stderr

C'est le flux que le terminal affiche lors d'une erreur. Pour ce flux il est possible de :

- l'envoyer dans un nouveau fichier : `cmd 2> file`
- l'envoyer à la fin d'un fichier : `cmd 2>> file`

stderr

C'est le flux que le terminal affiche lors d'une erreur. Pour ce flux il est possible de :

- l'envoyer dans un nouveau fichier : `cmd 2> file`
- l'envoyer à la fin d'un fichier : `cmd 2>> file`
- l'envoyez vers stdout : `cmd 2>&1 > file` ou `cmd &> file`

stderr

C'est le flux que le terminal affiche lors d'une erreur. Pour ce flux il est possible de :

- l'envoyer dans un nouveau fichier : `cmd 2> file`
- l'envoyer à la fin d'un fichier : `cmd 2>> file`
- l'envoyez vers stdout : `cmd 2>&1 > file` ou `cmd &> file`
- Donc dans ce dernier cas, l'ensemble de stderr et stdout sera dans *file*

stdin (exemple 1/2)

Il s'agit d'un flux invisible mais très utile.

Prenons par exemple le cas suivant : recherchez les lignes contenant *foo*

```
username@hostname:~$ cat liste.txt
# it displays the contents of list.txt
username@hostname:~$ cat liste.txt | grep foo
# it only prints the line containing foo
username@hostname:~$ cat liste.txt | grep ma_chaine > liste_filtre.txt
```

Ainsi la commande grep aura dans son stdin le contenu de *list.txt*, et elle pourra faire son traitement.

stdin (exemple 2/2)

Autre exemple de stdin : lire progressivement depuis le clavier

```
username@hostname:~$ sort << ENDSORT
> toto
> titi
> tata
> ENDSORT
tata
titi
toto
```

Le terminal attendra du texte et tant que la chaine de caractères est différente du mot-clé (ici ENDSORT), vous pouvez continuer. Dès que la chaine ENDSORT est détecté, toutes les données seront envoyées à `sort` . Le résultat `sort` sera envoyé dans sa sortie standard. On peut modifier le stdout :

```
username@hostname:~$ sort << ENDSORT > my_new_file_sort_from_keyboard
```

stderr (exemple)

```
username@hostname:~$ ls
File
Directory
username@hostname:~$ cat FileNotExist
cat: FileNotExist: No such file or directory
username@hostname:~$ cat FileNotExist 2> ./stderr.txt
username@hostname:~$ cat FileNotExist 2> ./stderr.txt 1> ./stdout.txt
username@hostname:~$ ls
File
Directory
stderr.txt
stdout.txt
username@hostname:~$ cat stderr.txt
cat: FileNotExist: No such file or directory
username@hostname:~$ cat stdout.txt

username@hostname:~$
```

ReGex

Les différents types

Il existe 2 types d'expressions régulières :

- expressions régulières basiques (vi, grep, expr, sed) : ERb

Les différents types

Il existe 2 types d'expressions régulières :

- expressions régulières basiques (vi, grep, expr, sed) : ERb
- expressions régulières étendues (grep -e, egrep, awk) : ERe

Les mots-clés des regex

- $\hat{=}$ Début de ligne

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- \$ = Fin de ligne

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- \$ = Fin de ligne
- . = N'importe quel caractère

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- \$ = Fin de ligne
- . = N'importe quel caractère
- [liste_de_caractères] = Un caractère parmi ceux listés

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- \$ = Fin de ligne
- . = N'importe quel caractère
- [liste_de_caractères] = Un caractère parmi ceux listés
- [$\hat{}$liste_de_caractères] = Un caractère qui n'est pas dans la liste

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- \$ = Fin de ligne
- . = N'importe quel caractère
- [liste_de_caractères] = Un caractère parmi ceux listés
- [$\hat{}$liste_de_caractères] = Un caractère qui n'est pas dans la liste
- * = 0 à n fois le caractère ou le groupe précédent

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- $ = Fin de ligne
- . = N'importe quel caractère
- [liste_de_caractères] = Un caractère parmi ceux listés
- [$\hat{}$liste_de_caractères] = Un caractère qui n'est pas dans la liste
- * = 0 à n fois le caractère ou le groupe précédent
- \ = Protection d'un caractère spécial

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- $ = Fin de ligne
- . = N'importe quel caractère
- [liste_de_caractères] = Un caractère parmi ceux listés
- [$\hat{}$liste_de_caractères] = Un caractère qui n'est pas dans la liste
- * = 0 à n fois le caractère ou le groupe précédent
- \ = Protection d'un caractère spécial
- PATERN{n} = Nombre de répétitions du motif (ERe)

Les mots-clés des regex

- $\hat{=}$ Début de ligne
- \$ = Fin de ligne
- . = N'importe quel caractère
- [liste_de_caractères] = Un caractère parmi ceux listés
- [$\hat{}$liste_de_caractères] = Un caractère qui n'est pas dans la liste
- * = 0 à n fois le caractère ou le groupe précédent
- \ = Protection d'un caractère spécial
- PATERN{n} = Nombre de répétitions du motif (ERe)
- (PATERN) = Groupe (ERe)

Exemple de ReGex (1/3)

Prenons ce fichier texte comme exemple :

toto likes titi

Isen 2021

Toto likes titi

Trouver les lignes qui commencent par toto et se terminent par titi

```
user@pem: cat exampleRegex.txt
toto likes titi
Isen 2021
Toto likes titi
user@pem: grep "^toto.*titi$" exampleRegex.txt
toto likes titi
```

Exemple de ReGex (2/3)

Trouver les lignes qui commencent par une majuscule

```
user@pem: cat exampleRegex.txt
toto likes titi
Isen 2021
Toto likes titi
user@pem: grep "^[A-Z]" exampleRegex.txt
Isen 2021
Toto likes titi
```

Exemple de ReGex (3/3)

Trouver les lignes qui contiennent “to” 2 fois

```
user@pem: cat exampleRegex.txt
toto likes titi
Isen 2021
Toto likes titi
user@pem: grep -E "(to){2}" exampleRegex.txt
toto likes titi
```

```
# SED

TODO: en -> fr

## Introduction

SED is a command that allows manipulation of text file from regular expression

By default sed displays the result in the stdout. To do the action directly in the
    file you need the option **-i**

## Substitution

Change one patern by another
```

Insert

Add a line before the wanted patern

```
sed "/PATERN_TO_LOOK_FOR/iPATERN_TO_ADD/"
```

```
user@pem: cat exampleRegex.txt
toto likes titi
Isen 2021
Toto likes titi
user@pem:  sed -e "/Toto/iTiti" exampleRegex.txt
toto likes titi
Isen 2021
Titi
Toto likes titi
```

Append

Add a line after the wanted patern

```
sed "/PATERN_TO_LOOK_FOR/aPATERN_TO_ADD/"
```

```
user@pem: cat exampleRegex.txt
toto likes titi
Isen 2021
Toto likes titi
user@pem:  sed -e "/toto/aTiti" exampleRegex.txt
toto likes titi
Titi
Isen 2021
Toto likes titi
```

Delete

Delete a line containing a pattern.

```
sed "/PATERN/d"
```

```
user@pem: cat exampleRegex.txt
toto likes titi
Isen 2021
Toto likes titi
user@pem: sed -e "/titi$/d" exampleRegex.txt
Isen 2021
```