Ecole Supérieure de Gestion, d'Informatique et des Sciences

Programmation système

(Scripts Shell et Python)

TRONC COMMUN
L3 IRT

Dr. Béthel ATOHOUN

Avril 2023

DÉFINITIONS

Qu’est-ce qu’un script ?

Une série de commandes définies dans un fichier pour l’automatisation d’une tâche donnée

Ces commandes sont exécutées les unes après les autres par un interpréteur (le shell dans notre cas)

Tout ce qui est réalisable en ligne de commande peut être inclus dans un script

Le **scripting** est une méthode pour l’automatisation de tâches

Cas d'utilisation

- **maintenance quotidienne d'un système**
- sauvegarde hebdomadaire de fichiers vers un serveur
- planification de tâches répétitives et/ou fastidieuses
- extraction de données utiles de fichiers volumineux
- etc.

RENDRE UN SCRIPT EXÉCUTABLE

Pour pouvoir exécuter un fichier de script, il faut définir dessus les droits de manière à ce que celui-ci devienne exécutable

Utilisation de chmod (change mode)

Syntaxe : chmod a+x nom_fichier

chmod a+x permet de rajouter le droit d'exécution à tous les utilisateurs pour le fichier nom_fichier

Exemple : chmod a+x monscript.sh

RENDRE UN SCRIPT EXÉCUTABLE

Les catégories d'utilisateurs	*Les permission*
u propriétaire (user) g groupe (group) o les autres (others) a tous (all)	r (4) : autorisation de lecture w (2) : autorisation d'écriture x (1) : autorisation d'exécution.

Exemple : **chmod a+x monscript.sh**

EXÉCUTER UN SCRIPT

Pour exécuter un script, on utilisera suivant le cas, un chemin relatif ou un chemin absolu

- **Chemin relatif :** Il faut être dans le dossier du fichier et faire

./nom_fichier

Exemple ./monscript.sh

Chemin absolu : Peut importe là où on se trouve, on précise le chemin allant de la racine jusqu'au fichier à exécuter

Exemple : /home/cark/monscript.sh en supposant que c'est le chemin absolu vers le fichier

ECRIRE UN SCRIPT

Editeur

On peut utiliser n'importe quel éditeur disponible dans l'environnement

Exemple : gedit, emacs, vi, vim, nano sous Linux ; notepad++ sous windows

Le Shebang

C'est la première ligne qu'on trouve dans un fichier de script shell

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

Il commence par un « # » (ce qui correspond la plupart du temps à des commentaires) puis d'un « ! » et enfin le chemin menant au programme Bash Shell

ECRIRE UN SCRIPT

Le Shebang

On y indique le chemin de l'interpréteur qui va lire le script.

Il en existe d'autres. Ci-dessous, quelques exemples :

```
#!/bin/sh

#!/bin/csh

#!/bin/zsh

#!/usr/bin/python

...
```

LES COMMENTAIRES

Les commentaires permettent de rendre des parties du contenu du fichier de script non interprétées par l'interpréteur

Une ligne commentée commence par le caractère #

On peut aussi commenter la fin d'une ligne

Mais il n'existe aucun moyen pour commenter plusieurs lignes à la fois.

Ainsi, en dehors du shebang, toute ligne commençant par # ne sera pas interprétée

```
#!/bin/bash
echo "Bonjour tout le monde"
#Ceci est un commentaire
echo "Merci pour l'explication" #Ceci ne sera pas interprété
```

Les variables

Variables = Espaces de stockage nommés

Il existe deux catégories de variables : Les variables définies par l'utilisateur et les variables prédéfinies (variables prépositionnées, variables d'environnement, etc.)

LES VARIABLES

Variables définies par l'utilisateur

Le nom d'une variable est un identificateur

- Caractères alpha-numériques
- Ne peut commencer par un chiffre
- Peut contenir le caractère *sous-tiret* (_)

Le nom d'une variable est sensible à la casse : une distinction est faite entre majuscule et minuscule

Par convention, il est en MAJUSCULE

Les variables

Syntaxe de déclaration

```
NOM_DE_LA_VARIABLE="valeur"
```

Nota : Il ne faut pas mettre d'espace entre le nom de la variable, le signe = et "

Syntaxe d'utilisation

Pour utiliser le contenu d'une variable, il faut faire précéder le nom de la variable par un $

Un exemple illustratif se trouve sur la page suivante

LES VARIABLES


```
#!/bin/bash
PRENOM="Luc"
NOM="SAGBO"
echo "Bonjour $PRENOM $NOM et bienvenue au cours de script shell"
```

#En supposant que le fichier du code ci-dessus se nome script.sh, et qu'on a déjà défini les droits d'accès appropriés dessus, son exécution se fera comme suit :

```
./script.sh
```

#ce qui donnera comme résultat :

Bonjour Luc SAGBO et bienvenue au cours de script shell

LES VARIABLES

Nota : Dans certains cas, on est obligé d'utiliser les accolades pour entourer le nom de la variable

```
#!/bin/bash
PRENOM="Luc"
NOM="SAGBO"
AGE="16"
echo "Bonjour $PRENOM ${NOM}, vous avez ${AGE}ans"
```

./script.sh
Bonjour Luc SAGBO, vous avez 16ans

Nota : Ici, si on ne mettait pas les accolades, on aurait de problèmes dans l'interprétation de **$NOM,** à cause de la virgule qui est collée au nom de la variable. De même on n'aurait pas pu afficher l'âge avec la chaine **ans** collée à la variable désignant l'âge.

Les variables

Assigner les sorties de commandes à une variable

Il est tout a fait possible d'assigner la sortie standard d'une commande à une variable en la mettant entre parenthèses

```
#!/bin/bash
PRENOM="Luc"
NOM="SAGBO"
MACHINE=$(hostname)
echo "Bonjour $PRENOM $NOM et bienvenu sur la machine ${MACHINE}."
```

LES VARIABLES

Assigner les sorties de commandes à une variable

Une autre syntaxe est également possible suivant le script et elle utilise le symbole ` (alt gr + 7) à la place du $() pour exécuter une commande.

```
#!/bin/bash
PRENOM="Luc"
NOM="SAGBO"
MACHINE=`hostname`
echo "Bonjour $PRENOM $NOM et bienvenu sur la machine ${MACHINE}."
```

Les variables spécifiques : prépositionnées

Quelques variables spécifiques au shell qui permettent de récupérer des informations sur les arguments passés ou sur l'exécution en cours.

$0 indique le nom d'une commande (la commande ou du script en cours d'exécution)

$# indique le nombre de paramètres positionnels d'une commande (nombre total d'arguments passés)

$$ indique le numéro de processus du shell courant (PID : Process ID)

$? Code qui indique un résultat lié au déroulement de la dernière commande exécutée

0 : Pas d'erreur. Autre valeur : cas contraire.

$* ensemble des paramètres (arguments) d'une commande (sous forme de liste séparée par le l'IFS)

$@ ensemble des paramètres (arguments) d'une commande (sous forme de liste séparée par des espaces)

$1 à $9 représentent des paramètres positionnels (les arguments passés à une commande ou à un script).

Les variables d'environnement

PATH représente les différents chemins où se trouvent les exécutables.

PS1 affiche la structure du *prompt* principal, ou celui par défaut.

HOME contient le répertoire de base de l'utilisateur.

SHELL indique le shell utilisé par l'utilisateur.

LANG indique la langue utilisée.

TEMP contient le chemin où les processus stockent les fichiers temporaires.

OSTYPE indique le type de système d'exploitation.

LD_LIBRARY_PATH contient les différents chemins où se trouvent les bibliothèques.

USER indique l'utilisateur connecté à la session courante.

EDITOR indique l'éditeur utilisé par défaut au sein de la session courante.

LES VARIABLES D'ENVIRONNEMENT

MANPATH	contient l'ensemble des répertoires où se situent les fichiers MAN.
MANPATH	indique le nom du fichier contenant l'historique des commandes enregistrées.
HISTSIZE	valeur limitant le nombre de commandes enregistrées dans le fichier historique.
PWD	contient le chemin du répertoire actuel.

Nota : La liste des variables spéciales et celles d'environnement présentées ici n'est que partielle. Pour connaitre par exemple toutes les variables d'environnement, il suffit de taper la commande env dans la console.

Vous êtes aussi priés de tester chacune des variables présentées sur les pages 15 à 17 pour mieux en comprendre l'utilité.

Pour tester une des variable, il suffit de faire echo $NOM_VARIABLE

Les Constantes : variables non modifiables

La notion de constante n'existe pas vraiment en Shell script.

Mais comme une constante n'est qu'une variable non modifiable, il y a la possibilité d'utiliser le mot clé **readonly** pour rendre une variable non modifiable

Syntaxe : readonly NOM_CONSTANTE="Valeur"

On vient donc de définir une variable comme une constante.

Les Tests

Lorsque vous tapez une commande, vous pouvez prendre le temps d'analyser la réponse du système et prendre une décision en fonction de cette réponse.

Il est tout à fait possible d'effectuer les mêmes étapes avec le scripting Shell grâce aux tests. Il existe deux formes de tests.

La syntaxe de la première forme, à base de crochets, est la suivante :

```
[ voici-la-condition-du-test-a-vérifier ]
```

Il est important de respecter les espaces après le symbole [mais également avant le].

Les Tests

Exemple 1 : Vérifier si le fichier /home/luc/test existe

```
[ -e /home/luc/test ]
```

- La commande retourne la valeur 0 (True) si le fichier existe
- La commande retourne la valeur 1 (False) si le fichier n'existe pas

Exemple 2 : Vérifier si la chaine contenue dans la variable NOM est vide ou non

```
[ -z NOM ]
```

- La commande retourne la valeur 0 (True) si la chaine est vide
- La commande retourne la valeur 1 (False) si la chaine n'est pas vide

Nota : Pour connaitre la valeur retournée par l'exécution d'une commande, il faut lire le contenu de la variable spéciale $? Après avoir exécuté la commande

Les Tests

Opérateurs principaux pour les tests

-e : 0 (True) si le fichier existe

-d : 0 (True) s'il s'agit d'un dossier

-r : 0 (True) si le fichier est disponible en lecture pour l'utilisateur

-s : 0 (True) si le fichier existe et n'est pas vide

-w : 0 (True) si le fichier est disponible en écriture pour l'utilisateur

-x : 0 (True) si le fichier est disponible en exécution pour l'utilisateur

-z : 0 si la chaîne de caractère est vide et 1 si elle ne l'est pas

–n : 1 si la chaîne de caractère est vide et 0 si elle ne l'est pas

Nota : Les deux dernières options s'utilisent sur les chaines de caractères

Les Tests

Comparer deux chaînes : Soit le script shell monscript.sh dont le code est ci dessous

```
#!/bin/bash
PRENOM='Luc'
NOM='SAGBO'
[ $PRENOM = $NOM ]
echo $?
```

```
./monscript.sh
 1
```

Nota : Avec =, le script nous renvoie 0 si les deux chaînes sont identiques et 1 si elles ne le sont pas

Les Tests

Comparer deux chaînes

```
#!/bin/bash
PRENOM='Luc'
NOM='SAGBO'
[ $PRENOM != $NOM ]
echo $?
```

```
./monscript.sh
 0
```

Nota : Avec !=, le script nous renvoie 0 si les deux chaînes ne sont pas identiques et 1 si elles le sont

Les Opérateurs

Les opérateurs permettent de faire des opérations sur les données et les valeurs.

❑ **Opérateurs arithmétiques**

Opérateur	Description	Exemple
+	Addition de valeurs	echo $((2+2))
*	Multiplication de valeurs	echo $((2*2))
-	Soustraction d'une valeur à une autre	echo $((4-2))
/	Divise des valeurs	echo $((4/2))
%	Renvoie le reste d'une division	echo $((10%3))

Nota : on peut aussi utiliser l'opérateur expr pour évaluer une expression.

Exemple : echo `expr 2 + 2`

Les Opérateurs

❑ Opérateurs logiques

Opérateur	Description	Exemple
&&	Correspond au ET logique	[5 -eq 5] && [6 -eq 7]
||	Correspond au OU logique	[5 -eq 5] || [7 -lt 5]

❑ Opérateurs de comparaison

Opérateur	Description
-eq	égale (comparaison numérique)
== ou =	égale (chaine de caractères)
-ne	différent (comparaison numérique)
!=	différent (chaine de caractère)

Opérateur	Description
-gt	Supérieur strict (comparaison numérique)
-ge	Supérieur ou = (comparaison numérique)
-lt	Inférieur strict (comparaison numérique)
-le	Inférieur ou = (comparaison numérique)

Les Opérateurs

- **Opérateurs d'affectation, d'incrémentation et associatifs**

Opérateur	Description	Exemple
++	incrémentation	echo $((var++)) $var
--	décrémentation	echo $((var--)) $var
=	affectation	echo $((var=4))
*=	Stocke dans une variable son produit par une valeur	echo $((var*=4))
+=	Augmente une variable d'une valeur	echo $((var+=4))
-=	Diminue une variable d'une valeur	echo $((var-=4))
/=	Stocke dans une variable sa division par une valeur	echo $((var/=4))
%=	Stocke dans une variable le reste de sa division par une valeur	echo $((var%=4))

Les Opérateurs

Opérateur	Description	Exemple
>	Envoie le flux de sortie à un fichier.	ps > processes.txt
>>	Ajoute du contenu à un fichier déjà existant. Crée le fichier s'il n'existe pas.	ps -ax >> processes.txt
<	Lit l'entrée standard à partir d'un fichier.	grep word < logs.txt
2>	Redirige le flux d'erreurs vers un fichier	ps 2> errors.txt
2>>	Redirige et ajoute le flux d'erreurs vers un fichier existant. Crée le fichier s'il n'existe pas	ps 2>> errors.txt

ENTRÉE UTILISATEUR

La commande read permet d'accepter les données venant de l'entrée standard (STDIN). L'utilisateur peut dont entrer des données à partir du clavier.

Syntaxe d'utilisation

La syntaxe de base est la suivante : read NOM_VARIABLE

Lorsqu'on souhaite afficher un message indicatif avant la lecture, on peut utiliser l'option –p

read –p "Le message indicatif : " NOM_VARIABLE

Exemple :

```
read –p "Entrez votre nom : "  NOM
```

Ce qui est équivalent à :

```
echo "Entrez votre nom : "
read NOM
```

Entrée Utilisateur

On peut lire dans plusieurs variables à la fois avec une seule instruction read

Exemple : read NOM PRENOM

Nota : lorsque à la saisie, l'utilisateur entre :

- moins de valeurs que de variables, les dernières variables sont vides
- plus de valeurs que de variables, la dernière variable prend le reste des valeurs, séparateurs compris

Le séparateur par défaut est soit le caractère espace, soit tabulation (\t) ou soit saut de ligne (\n). Mais on peut définir un autre séparateur.

On change pour cela la valeur par défaut de la variable prédéfinie IFS

Entrée Utilisateur

Exemple :

```
cark@osboxes:~$ OLDIFS="$IFS"
cark@osboxes:~$ IFS="|"
cark@osboxes:~$ read A B C
azerty|qwerty|papou
cark@osboxes:~$ echo $B
qwerty

cark@osboxes:~$ IFS="$OLDIFS"
cark@osboxes:~$ read A B C
azerty|qwerty|papou
cark@osboxes:~$ echo $A
azerty|qwerty|papou
```

Entrée Utilisateur

On a également la possibilité d'utiliser le commande **read** pour lire une donnée sans l'afficher lors de la saisie.

Ceci peut-être très utile pour la saisie de mot de passe.

Pour cela on ajoute l'option –s à l'option –p.

Nota : avec l'option **-s**, il est recommandé d'ajouter la commande **echo** pour afficher un saut de ligne.

```
cark@osboxes:~$ read -p "Login ? " LOGIN
cark@osboxes:~$ read -s -p "Mot de Passe ? " PASSWD; echo
```

Les Structure de contrôle

Il y a deux catégories de structures de contrôle :

- Les Structure conditionnelles
  - La structure **if** avec toutes ses variantes
  - La structure **case**
- Les structures répétitives ou itératives
  - La boucle **for**
  - La boucle **while**
  - la boucle **until**

Les Structure de contrôle

Les conditions : le cas de if

1er Cas

```
if [ condition-est-vraie ]
then
        command1
        command2
fi
```

2nd Cas

```
if [ condition-est-vraie ]
then
        command1
else
        command2
fi
```

3ème Cas

```
if [ condition1-est-vraie ]
then
        command1
elif [ condition2-est-vraie ]
then
        command2
else
        command3
fi
```

Les Structure de contrôle

Exercice :

Créer un script qui demande à l'utilisateur de saisir une note comprise entre 0 et 20 et qui affiche un message en fonction de cette note :

"très bien" si la note est entre 16 et 20, 16 inclus ;

"bien" lorsqu'elle est entre 14 et 16, 16 exclue ;

"assez bien" si la note est entre 12 et 14, 14 exclue ;

"passable" si la note est entre 10 et 12 ; 12 exclue

"insuffisant" si la note est inférieur à 10.

Les Structure de contrôle

Correction Exercice :

```
#!/bin/bash

echo "Entrez votre note :"
read note
if [ "$note" -ge 16 ]; then
        echo "très bien"
elif [ "$note" -ge 14 ]; then
        echo "bien"
elif [ "$note" -ge 12 ]; then
        echo "assez bien"
elif [ "$note" -ge 10 ]; then
        echo "moyen"
else
        echo "insuffisant"
fi
```

Les Structure de contrôle

Les conditions : le cas de case

Syntaxe

```
case <valeur> in
[PATTERN [I PATTERN] )
    <liste d'instructions>
    ;;]
esac
```

Syntaxe

```
case "$VARIABLE" in
        premier_cas)
                commande1
                ;;
        deuxieme_cas)
                commande2
                ;;
        troisieme_cas)
                commande3
                ;;
esac
```

Exemple

```
case "$1" in
    start)
        `/etc/init.d/apache2 start`
        ;;
    stop)
        kill $(cat /var/run/apache2/apache2.pid)
        ;;
      *)
        echo « Indiquez start ou stop«
        exit 1
        ;;
esac
```

Les Structure de contrôle

Les Boucles : la boucle for

Syntaxe d'utilisation

```
for VARIABLE in OBJET1 OBJET2 OBJET3 OBJETn
do
        command1
        command2
done
```

Les Structure de contrôle

Les Boucles : la boucle for

Exemple :

```
#!/bin/bash

for CHIFFRE in 10 11 12 13
do
        echo "Chiffre : $CHIFFRE"
done
```

```
#!/bin/bash

for CHIFFRE in {10..13}
do
        echo "Chiffre : $CHIFFRE"
done
```

Exercice

Créer un script qui prend un nombre en saisie et l'élève à sa propre puissance. C'est un peu le même principe que la factorielle mais cette fois, **l'usage de la boucle for est imposé**.

Les Structure de contrôle

Les Boucles : la boucle while

Syntaxe d'utilisation

```
while [ la-condition-est-vraie  ]
do
        command1
        command2
done
```

Les Structure de contrôle

Les Boucles : la boucle while

Exemple :

```
#!/bin/bash
while [ -z $PRENOM ]
do
        read -p "Quel est votre prenom ?" PRENOM
done
echo "Votre prenom est $PRENOM"
```

Exercice

Création d'un jeu qui permet à l'utilisateur de deviner un chiffre généré par le script entre 1 et 50. A chaque fois que l'utilisateur entre un chiffre, le script lui indique si le chiffre à trouver est supérieur ou inférieur à celui qu'il a entré, etc.

Les Structure de contrôle

Les Boucles : la boucle until

La boucle until fait la même chose que la boucle while, à la différence qu'ici la boucle est exécutée tant que la condition est fausse (contrairement à la boucle while)

```
#!/bin/bash
until [ -n "$PRENOM" ]
do
    read -p "Quel est votre prenom ?" PRENOM
done
echo "Votre prenom est $PRENOM"
```

Les Fonctions

Une fonction est un block de code désigné sous un nom et spécialisé dans un traitement. Les fonctions présentent plusieurs avantages, à savoir :

- **écrire un block de code une fois, et le réutiliser autant de fois que nécessaire**
- **avoir du code modulaire et aéré**
- **faire facilement la maintenance du code**

Il existe deux syntaxe de définition d'une fonction. Les deux sont présentées sur la page qui suit :

Les Fonctions

#utilisation du mot clé "function"

```
function nom_de_la_fonction(){
        command1
        command2
}
```

#non utilisation du mot clé "function"

```
nom_de_la_fonction(){
        command1
        command2
}
```

Les Fonctions

Appel d'une fonction

On se sert de son nom sans toutefois le faire suivre des parenthèses

```
#!/bin/bash
function check_server() {
        ping -c 1 192.168.8.1   #on suppose que l'IP du serveur est 192.168.8.1
        if [ $? -eq 0 ] then ;
                echo "Le serveur est joignable"
        else
                echo "Le serveur n'est pas joignable"
        fi
}
```

Les Fonctions

Appel d'une fonction

```
#!/bin/bash
#suite du code de la page précédente : appel de la fonction
check_server      #ceci est l'appel de la fonction
```

```
PING 192.168.8.1 (192.168.8.1) 56(84) bytes of data.
64 bytes from 192.168.8.1: icmp_seq=1 ttl=63 time=2.51 ms
--- 192.168.8.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.508/2.508/2.508/0.000 ms
serveur joignable
```

Les Fonctions

Passage de paramètres à une fonction

les fonctions peuvent accepter des paramètres comme d'ailleurs c'est le cas avec les scripts en général.

Les N paramètres sont accessibles à travers les variables spéciales $1, $2, …, $n. Naturellement, $0 fait référence au nom du script lancé.

```
#!/bin/bash
   function check_server() {
            ping -c $2 $1   #ici $2 représente le nombre de paquets à lancer, $1 représente l'adresse IP à atteindre
            if [ $? -eq 0 ] then ;
                        echo "Le serveur est joignable"
            else
                        echo "Le serveur n'est pas joignable"
            fi
   }
```

Les Fonctions

Appel d'une fonction

```
#!/bin/bash
#suite du code de la page précédente : appel de la fonction
check_server "192.268.8.1"  "1"  #ceci est l'appel de la fonction en lui passant les paramètres
```

```
PING 192.168.8.1 (192.168.8.1) 56(84) bytes of data.
64 bytes from 192.168.8.1: icmp_seq=1 ttl=63 time=2.51 ms
--- 192.168.8.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.508/2.508/2.508/0.000 ms
serveur joignable
```

Les Fonctions

Variables Locales et Globales

Les variables locales ne sont visibles et accessibles qu'à l'intérieur de la fonction.

Par contre une variable globale est accessible à l'intérieur d'une fonction. Mais pour cela il faudra la déclarer avant la fonction.

On peut néanmoins déclarer une variable globale dans une fonction. Dans ce cas, elle ne sera accessible qu'une fois la fonction appelée.

Pour déclarer une variable locale on doit précéder le nom de la variable du mot clé **local**

Les Fonctions

Variables Locales et Globales : Exemple

```
#!/bin/bash
VAR1="Globale1"
function test_variables() {
   echo "La variable $VAR1 est accessible"
   VAR2="Globale2"
   local VAR3="Locale1"
   echo "variables $VAR2 et $VAR3"
}
test_variables
echo "les variables $VAR1 et $VAR2 et $VAR3"
```

En exécutant le code, on se rend compte que toutes les variables sont disponibles dans la fonction. Mais une fois en dehors de la fonction, la variable $VAR3 n'est plus accessible.

Je vous propose de tester ce code et de voir le résultat.

Les Wildcards : caractères génériques

Un wildcard est un caractère (ou une chaîne de caractères) générique ou de substitution qui est utilisé dans une expression pour prendre un certain nombre de valeurs

On les utilise avec la plupart des commandes, mais très souvent avec les commandes ls, mkdir et rm

Caractères spéciaux : ? et *

? Désigne exactement un seul caractère

Exemple

```
$ echo fichier?.txt      #fichiers fichier1.txt, fichier2.txt, fichiers.txt, si ces fichiers existent dans le dossier courant
$echo t?t?.txt           #fichiers toto.txt, tato.txt, titi.txt, si des fichiers comme ça existent dans le dossier courant
```

Les Wildcards : caractères génériques

* Désigne n'importe quel caractère ou suite de caractères, y compris le caractère vide

Exemple

```
$ echo *t*      #peut donner table, fictif.txt, toto.txt, toise, si ces fichiers ou dossiers existent
$ echo t*.txt   #tous les fichiers txt commençant par t, s'ils existent
```

Nota : On ne peut pas utiliser ? ou * au début pour retrouver les fichiers préfixés par le caractère point (.)
Il faut le spécifier explicitement avant de mettre ? ou *

Exemple

```
$ echo .*
```

Les Wildcards : caractères génériques

Les classes de caractères :

classe	signification
[abcd]	L'un des caractères a, b, c ou d
[!abcd]	Tout caractère sauf les caractères a, b, c et d
[c-g]	L'un des caractères de l'alphabet de c à g
[1-6]	L'un des chiffres de 1 à 6
[[:alpha:]]	Toutes les lettres de l'alphabet (minuscules et majuscules)
[[:alnum:]]	Lettres de l'alphabet ainsi que chiffres (0 à 9)
[[:digit:]	Tous les chiffres de 0 à 9
[[:space:]]	Tous caractères d'espacement (espace, tabulation, etc…)
[[:upper:]]	Toutes les lettres de l'alphabet en majuscule
[[:lower:]]	Toutes les lettres de l'alphabet en minuscule

Les Filtres

Les filtres sont des commandes qui peuvent être appliqués sur les fichiers textes ou sur les résultats d'une commande produisant du texte. Les filtres, comme leur nom l'indique, permettent de filtrer (restreindre, épurer, clarifier) un flux de données.

Les commandes qui lisent et écrivent sur les E/S sont appelées des filtres. Les filtres sont souvent utilisés en branchement de commandes (en réalisant un tube avec l'aide du caractère | ou caractère pipe).

Les Filtres

Les commandes head et tail

head et **tail** permettent respectivement de n'afficher que les premières ou dernières lignes d'un fichier ou de l'entrée standard.

syntaxe :

head [options] <fichier> tail [options] <fichier>

Exemples :

$ head -n 15 /var/log/syslog

$ cat /var/log/syslog | tail -n 15

Les Filtres

Les commandes head et tail

head et **tail** permettent respectivement de n'afficher que les premières ou dernières lignes d'un fichier ou de l'entrée standard.

syntaxe :

head [options] <fichier> tail [options] <fichier>

Exemples :

$ head -n 15 /var/log/syslog

$ cat /var/log/syslog | tail -n 15

LES FILTRES

- La commande split
- La commande split permet de découper un fichier en plusieurs fichiers plus petits (si possible de même taille).
- Les fichiers créés par la commande split seront nommés PREFIXEaa, PREFIXEab, …, la chaîne PREFIXE étant donnée en argument de la commande.

Exemples :

```
$ split -b 22M /var/log/syslog log_      #l'option –b permet de spécifier la taille des fichiers log_
split -l 30 /var/log/syslog log_         #l'option –l permet de spécifier le nombre de lignes
split -n 5 /var/log/syslog log_          #l'option –n permet de spécifier le nombre de fichiers
```

LES FILTRES

La commande cut

C'est une commande qui permet d'extraire des colonnes d'un fichier.

Syntaxe : cut <options> … <fichier> …

L'option –b permet de préciser le nombre et les positions des caractères à extraires

L'option –f permet de préciser les champs à extraire

Nota : Le séparateur par défaut est la tabulation (ce qui est différent de l'espace). Lorsqu'on veut définir son propre séparateur, on utilise l'option -d

Pour plus de détail, consulter l'aide avec la commande man cut

LES FILTRES

La commande cut

C'est une commande qui permet d'extraire des colonnes d'un fichier.

Exemples :

$ cut -c1-2 tri.txt #extrait les deux premiers caractères du fichier tri.txt

$ cut -f2,5 tri.txt #extrait les seconde et cinquième colonnes

$ cut –f1 –d " " .bash_history #première colonne

$ cat extr.txt | cut -d " " -f 2

LES FILTRES

Autres filtres : filtres puissants

- find
- grep
- awk
- sed

Il faut se documenter pour comprendre le fonctionnement de ces filtres. « man » est votre ami.