

**Linux shell  
bash (partie 1)**

Formation continue  
Collège de Maisonneuve

2030, boul. Pie IX, bureau 430

Montréal (Québec) H1V 2C8

Rédacteurs  
Hakim Benameurlaine – Juin 2016  
Louis Savard – Janvier 2023

Table des matières

[Introduction 4](#_Toc124853666)

[BASH 8](#_Toc124853667)

[Le shell 8](#_Toc124853668)

[Démarrage du *shell* 8](#_Toc124853669)

[Les scripts de connexion 8](#_Toc124853670)

[Personnalisation des commandes bash 9](#_Toc124853671)

[Personnalisation du login utilisateur 9](#_Toc124853672)

[Facilités de saisie des commandes 9](#_Toc124853673)

[Historique des commandes 9](#_Toc124853674)

[Le clic-droit 9](#_Toc124853675)

[L'opérateur tilde (~) 9](#_Toc124853676)

[Complétion d’une commande 9](#_Toc124853677)

[Ensemble de fichiers 10](#_Toc124853678)

[Caractères spéciaux 11](#_Toc124853679)

[Guillemets (*double quote*) 11](#_Toc124853680)

[Guillemets simples (*single quote*) 12](#_Toc124853681)

[Backslash (\) 12](#_Toc124853682)

[Apostrophe inversés (*backquote*) 13](#_Toc124853683)

[Ligne de commandes 13](#_Toc124853684)

[Analyse de la ligne de commande 13](#_Toc124853685)

[Enchaînement inconditionnel des commandes 14](#_Toc124853686)

[Enchaînement conditionnel des commandes 14](#_Toc124853687)

[Redirections des entrées-sorties 15](#_Toc124853688)

[Substitution de commande 23](#_Toc124853689)

[Programmation bash 24](#_Toc124853690)

[Introduction 24](#_Toc124853691)

[Création d’un script 24](#_Toc124853692)

[Exécution du script 25](#_Toc124853693)

[Mise au point (débogage) 25](#_Toc124853694)

[Entrées-Sorties 26](#_Toc124853695)

[Les variables bash 27](#_Toc124853696)

[Variables utilisateur 27](#_Toc124853697)

[Assignation d’une valeur à une variable 28](#_Toc124853698)

[Substitution de variable 28](#_Toc124853699)

[Variables exportées 29](#_Toc124853700)

[Opérateur {} 29](#_Toc124853701)

[Variables d'environnement 30](#_Toc124853702)

[Variables système 31](#_Toc124853703)

[Paramètres de position 32](#_Toc124853704)

[Commandes importantes 32](#_Toc124853705)

[shift 32](#_Toc124853706)

[Tr 32](#_Toc124853707)

[set 33](#_Toc124853708)

[eval 33](#_Toc124853709)

[basename 34](#_Toc124853710)

[seq 35](#_Toc124853711)

[test 36](#_Toc124853712)

# Introduction

L’interpréteur de commandes (*shell*) **permet d'interagir directement avec un système Linux** …   
on peut énoncer des **commandes qui sont évaluées et exécutées** par l’interpréteur de commandes, généralement au moyen du démarrage de programmes externes.

En outre, la plupart des interpréteurs de commandes incluent des **fonctionnalités de langage de programmation**.  
Ils permettent …

* la création des **variables** ;
* la mise en place de **structures de contrôle** telles que   
  des **conditions**, des **boucles**, des **fonctions** ;
* **...**

Ainsi, il est possible de placer des **séquences complexes de commandes shell** et des **appels de programmes externes** dans des **fichiers texte** et les **faire interpréter comme des scripts shell**.   
Cela rend les opérations difficiles répétables et les processus se reproduisent sans effort.

Le système Linux utilise des **scripts shell pour de nombreuses tâches internes**.   
Par exemple, les opérations d’ouverture et de fermeture de session sont généralement implémentés sous forme de scripts shell.   
Cela s'applique également à de nombreuses commandes système.   
Linux permet de cacher le fait qu’un programme système n’est pas un code machine directement exécutable mais un script shell à ses utilisateurs, du moins en ce qui concerne la syntaxe d’appel

Un script est une **suite d’instructions**, de commandes qui constituent un **scénario d'actions**.   
C’est un **fichier texte que l’on peut exécuter**, c’est à dire, **lancer comme une commande**.

Un **script shell est un langage de programmation à part entière**. Il peut définir des variables, des fonctions et il est également possible d’effectuer une exécution conditionnelle de commandes shell.

Bash (AKA **B**ourne **A**gain **Sh**ell) est un type d'interpréteur qui traite les commandes shell.

Un interpréteur shell prend des commandes au format texte brut et appelle les services du système d'exploitation pour faire quelque chose.

Par exemple, la commande ls répertorie les fichiers et dossiers dans un répertoire.

Bash est la version améliorée de sh (Bourne Shell).

Un script shell écrit un programme à exécuter par le shell et un script shell est un fichier ou un programme que le shell exécutera.

Un script shell est donc une **séquence de commandes pour laquelle on a une utilisation répétée**.

Cette séquence est généralement **exécutée en entrant le nom du script sur la ligne de commande**.

On peut également utiliser des scripts pour **automatiser les tâches à l'aide de la fonction cron**.

Une autre utilisation des scripts est la procédure de démarrage et d'arrêt Linux, où le fonctionnement des démons (*daemons*) et des services est défini dans les scripts d'initialisation.

L’écriture de scripts shell peut répondre à plusieurs besoins différents …

* Les **scripts permettent simplement de lancer des commandes disposant d’options complexes**, et qu’on désire employer régulièrement ;
* Les **scripts d’initialisation**, ou de **démarrage** (boot), du système, permettent d’**identifier les périphériques** **disponibles et de les initialiser**, de **préparer les systèmes de fichiers** **et les partitions**, et de **configurer** **les communications** **et les** **services réseau**.   
  Ils sont normalement livrés avec le système d’exploitation, mais l’administrateur expérimenté doit parfois intervenir dans ces scripts (souvent longs et fastidieux) pour **personnaliser une station de travail ou un serveur** ;
* Certaines **applications requièrent une configuration complexe**, et des scripts shell sont employés afin de s’**assurer la gestion de configurations**, la **préparation des répertoires** et des **fichiers temporaires**, … ;
* La **supervision d’un parc informatique implique de nombreuses tâches automatisées** afin de vérifier l’état des systèmes (utilisation de la mémoire et des espaces disque, antivirus...) et assurer des tâches de maintenance périodique (effacement des fichiers temporaires, rotation des fichiers de journalisation, ...) ;
* Plusieurs serveurs utilisent une **interface constituée de scripts shell afin de transmettre des requêtes** SQL au logiciel de base de données (Oracle, MySQL, ….).

## Écriture d’un script

Les scripts shell ne sont en réalité que des **séquences de commandes shell** qui ont été **stockées dans un fichier texte**.

Le shell peut prendre ses **entrées depuis le clavier** (entrée standard) ou une autre source, comme un **fichier de script** **shell** - il n'y a **aucune différence en ce qui concerne l'exécution des commandes**.   
Par exemple, les sauts de ligne servent de séparateurs de commandes, un peu comme sur la vraie ligne de commande.

### **Quelques conseils de lisibilité**

Bien que l’on puisse, pour plus de commodité, saisir quelque chose comme …  
**⟨commande1⟩ ; ⟨commande2⟩**sur la ligne de commande, il est **préférable d’écrire une commande par ligne dans un script**, par souci de clarté …

**⟨commande1⟩**

**⟨commande2⟩**

Il n'y a aucune différence en ce qui concerne l'exécution des commandes.

Il est encore possible d’**augmenter** **la lisibilité d'un script en utilisant judicieusement des lignes vides**, qui sont **ignorées** **par les shells**, tout comme sur la ligne de commande.

Le **shell ignore également tout ce qui suit un diese ( # )**.   
Cela permet d’ajouter les scripts de commentaires …

**# Commande1 est exécutée en premier**

**⟨commande1⟩**

**⟨commande2⟩ # C’est la seconde commande**

Les **scripts plus longs doivent commencer par un bloc de commentaires** contenant le nom du bloc de commentaires, son but et son fonctionnement, comment l’appeler, …   
Le **nom de l’auteur et un historique des versions** peuvent également y figurer.

**Remarque** ...  
Les **fichiers texte marqués comme exécutables** à l'aide de la commande chmod sont considérés, par défaut, comme des **scripts pour le shell /bin/sh** - sur les systèmes Linux, c'est souvent (mais pas toujours) un **synonyme de bash**.

### Un premier script

Pour débuter, on écrit un fichier de texte simple. Pour ce faire, n’importe quel éditeur de texte suffira à la tâche.

**#!/bin/bash**

**clear**

**echo "Le script démarre maintenant"**

**echo "Bonjour $USER"**

**echo**

**echo "Voici une liste des usagers présentement connectés"**

**echo**

**w**

**echo**

**echo "On définit deux variables ..."**

**couleur="bleu"**

**valeur="9"**

**echo "Ceci est une chaine de caractères ... $couleur"**

**echo "Et voici une valeur ... $valeur"**

**echo**

**echo "Retour au prompt ...**

**Conseil** …

Il suffit de saisir le texte dans un éditeur de texte.   
Il peut être judicieux de créer un répertoire /var/scripts afin de classifier les scripts.   
Par la suite, il est possible de d’ajouter le répertoire au contenu de la variable PATH …  
**>>** **export PATH="$PATH:/var/scripts"**

Il faudra modifier les permissions d’accès à ce répertoire afin de donner le droit d’écriture aux utilisateurs concernés.

## Exécution du script

Il existe **deux façons d’exécuter une fichier texte comme un script** …

* On appelé l’interpréteur de commande pour exécuter le fichier …  
  **>>** **bash script.sh**  
  Dans cet exemple, bash est l’interpréteur de commande ;
* Le **script doit avoir des autorisations d'exécution pour les propriétaires corrects** afin d'être exécutable.   
  Lors de la définition des autorisations, on vérifie que l’on a vraiment obtenu les autorisations souhaitées. Lorsque cela est fait, le **script peut s'exécuter comme n'importe quelle autre commande** …  
  **>>** **chmod u+x script.sh**  
  **>>** **ls -l script.sh**  
  **>> script.sh**

Le shell spécifié dans un script démarre en tant que sous-shell du shell actuel et exécute le script.   
Cela est fait lorsque l’on souhaite que le script démarre avec des **options spécifiques ou dans des conditions spécifiques** qui ne sont pas spécifiées dans le script.

Si on **ne veut pas démarrer un nouveau shell mais exécuter le script dans le shell actuel**, on doit utiliser la commande source …

**>>** **source script.sh**

**Remarque** …  
La source intégrée Bash est un synonyme du shell Bourne **./**.

Le script n'a pas besoin de l'autorisation d'exécution dans ce cas.   
Les commandes sont exécutées dans le contexte du shell actuel, donc toutes les modifications apportées à son environnement seront visibles lorsque le script aura terminé son exécution.

# Bash

## Le shell

Un interpréteur de commandes est un programme qui sert d'**intermédiaire entre l'utilisateur et le système d'exploitation**.

Son essentielle tâche est l'exécution de commandes.

### Démarrage du *shell*

Suit au chargement du système d’exploitation l’interpréteur de commandes prend la relève (*shell*) …

* Lors de la **création d’un compte usager**, un **shell lui est automatiquement associé** ;
* Dans le **fichier /etc/passwd**, le **dernier champ** contient le nom du *shell* associé à l’usager correspondant   
  (par **défaut /bin/bash** sur Debian et CentOS) ;
* Le **shell associé est ainsi lancé automatiquement** dès l’ouverture de la session ;
* Le **shell exécute les scripts globaux à tous les utilisateurs** et,   
  par la suite,   
  les **scripts liés au compte qui** permettent une personnalisation.
* Enfin, il affiche **l’invite de commandes** (*prompt*) et se met en attente de la **lecture d'une commande** ;
* La **commande** exit permet de **quitter le shell**.

### Les scripts de connexion

* Le **script global /etc/profile** est **exécuté pour tous les usagers**,   
  y compris pour l’usager **root** ;
* Ce dernier cherche à exécuter tous les **scripts /etc/profile.d/\*.sh** ;
* Par la suite, il y a **exécution du script personnel $HOME/.bash\_profile**   
  (la variable $HOME représente le chemin vers le répertoire personnel).   
  Il s'agit d'un **fichier de démarrage personnel et paramétrable** ...
  + **Si .bash\_profile n’existe pas** alors il cherche à exécuter **.bash\_login**.
  + **Si .bash\_login n’existe pas** à son tour alors il cherche à exécuter **.profile**.
* À son tour, il exécute **$HOME/.bashrc** dans lequel il est recommandé de placer toutes les **fonctions ou alias** **personnels** (car **.**bashrc **est exécuté pour tout les shells**) ;
* Enfin le précédent script exécute /**etc/bashrc, dans lequel on place les alias globaux** et la **définition symbolique du prompt $PS1** ;
* En dernier lieu,   
  l’invite de commmande (*prompt*) utilisateur s'affiche et le shell attend une commande ;
* Le **script $HOME/bash\_logout** est exécuté lorsque la **session est terminée** (**exit** ou **CTL-D**).

### Personnalisation des commandes bash

**/etc/bashrc** étant le dernier script d'initialisation du *shell* bash, root peut y définir des alias globaux pour tous les usagers.

>> nano /etc/bashrc

alias effacer\_ecran=clear

### Personnalisation du login utilisateur

Chaque usager peut ajouter des commandes shell au fichier de profil personnel, **~/.bash\_profile.**Voici un exemple …

clear

salut="Bonjour $USER ! Nous sommes le $(date)"

echo $salut

## Outils de saisie des commandes

Comme les commandes Linux sont souvent longues à saisir, diverses facilités sont offertes …

### Historique des commandes

Cette liste numérotée est accessible en tapant:

>> history | less

Pour relancer la commande numéro **n**, saisir (sans espace) …

>> !n

On peut aussi parcourir les précédentes lignes de commandes avec les flèches directionnelles et les éditer.

Ceci permet très facilement de reprendre une précédente commande afin de l'éditer et de la modifier.

### Le clic-droit

Dans un terminal console, à la suite d’une sélection de texte quelconque, un clic droit recopie ce texte sur la ligne de commande.

### L'opérateur tilde (~)

Le caractère tilde ~ (ALT + 126) **seul** renvoie au **répertoire personnel** de l'usager actuel.

Le tilde **~** suivi d'un nom d'usager, par exemple **oracle**, renvoie au **répertoire personnel** de l’usager **lsavard**.

>> cd ~lsavard

### Complétion d’une commande

Dans un terminal, lors de la saisie d’une commande, la touche TAB cherche à compléter la commande ou le nom du fichier.

S'il y a plusieurs propositions, il y a attente d'un complément d'informations de la part de l'usager (avec un **beep** sonore). En appuyant une seconde fois sur la touche **TAB**, l'interpréteur de commandes affiche toutes les possibilités où en indique le nombre (s'il y en a beaucoup).

### Ensemble de fichiers

Travailler avec le shell nécessite souvent de **manipuler des ensembles de fichiers**.

L'utilisation de caractères spéciaux (appelés aussi **méta-caractères**) dans les noms de fichiers, permet de générer des modèles pour désigner ces ensembles.

Il existe quatre constructeurs de modèles …

|  |  |
| --- | --- |
| **Modèle** | **Signification** |
| **\*** | Remplace une chaîne de longueur quelconque, même si elle est vide |
| **?** | Remplace un seul caractère quelconque |
| **[ ]** | Remplace un caractère quelconque de la liste ou de l'intervalle |
| **[^   ]** | Remplace n'importe quel caractère **sauf** ceux de la liste |

Le modèle **[ ]** permet de sélectionner un élément de la liste ou de l'intervalle spécifié.

Dans le cas d'une suite ordonnée de caractères comme abc...z, on peut utiliser la notation intervalle **a-z**

On peut mélanger les deux notations, comme dans [a-z].[0-9], ensemble des fichiers a.0, a.1, .., b.0 b.1 …

**Exemples**

**Commande (s)**

>> ls a\*

**Résutltat (s)**

Liste les fichiers du répertoire courant dont le nom commence par a (peu importe le nombre de caractèeres qui suivent).  
Liste également les répertoires.

**Commande (s)**

>> ls [a-d, A-D]\*

**Résutltat (s)**

Liste les fichiers du répertoire courant dont le nom commence par a, b, c ou d minuscule ou majuscule (peu importe le nombre de caractèeres qui suivent).  
Liste également les répertoires.

**Commande (s)**

>> cp prog1[000-900] /home/java

**Résutltat (s)**

Copie tous les fichiers prog1000 jusqu'à prog1900 dans le répertoire /home/java.

## Caractères spéciaux

|  |  |
| --- | --- |
| Caractères | Explication |
| **$** | Indique une variable shell |
| **|** | Rediriger la sortie standard vers la prochaine commande |
| **#** | Indique un commentaire |
| **&** | Exécute un processus en arrière-plan |
| **?** | Remplace un seul caractère quelconque |
| **\*** | Remplace une chaîne de longueur quelconque, même vide |
| **>** | Redirige vers une autre sortie (qui est habituellement l’écran) |
| **<** | Redirige l’entrée |
| **`** | Substitue une commande (*backquote*) – on peut également utiliser **$(commande)** |
| **>>** | Redirige vers une autre sortie sans écraser le contenu existant (qui est habituellement l’écran) |
| **[ ]** | Indique une liste ou un intervalle |
| **[a-z]** | Indique un intervalle |
| **[a,z]** | Indique une liste |
| **. nom\_fichier** | Exécute le fichier *nom\_fichier* dans le même shell |
| **espace** | Sépare les différents éléments d’une commande |
| **"** | Indique une action spécifique (guillemets – *double quote*) |
| **‘** | Indique une action spécifique (apostrophe – *single quote*) |
| **\** | Insère un caractère qui est interprété comme tel (barre oblique inversée – *backslash*) |
| **;** | Sépare une séquence de plusieurs commandes |

### Guillemets (*double quote*)

Habituellement, une ligne de commande saisie à l’invite de commande (*prompt*) de la console ou bien inscrite dans un script est une **phrase composée de mots séparés par des espaces** (ou des **tabulations**).

* Le **premier mot** est considéré comme le **nom d'une commande** et le shell cherche à l’exécuter ;
* Les **mots suivants** sont des **options** ou des **paramètres pour cette commande**.

Afin d’inhiber cette **interprétation des espaces**, il faut entourer le **groupe de mots de "guillemets"**, ce groupe sera alors interprété comme **un seul** paramètre.

**Exemple**

Afin de rechercher la chaîne Louis Savard (qui constitue un seul paramètre) sur les lignes de **/etc/passwd**, il est possible de formuler la commande de ma manière suivante …

**Commande (s)**

>> grep -i "root" /etc/passwd

**Résultat (s)**

root:x:0:0:root:/root:/bin/bash

**Commande (s)**

>> var="bonjour"

>> echo $var

**Résultat (s)**

bonjour

**Commande (s)**

>> newvar="La valeur de var est … $var."

>> echo $newvar

**Résultat (s)**

La valeur de var est … bonjour.

L'option -i de la première commande permet de s'affranchir de la casse (MAJUSUCLE ou minuscule).

### Guillemets simples (*single quote*)

L’utilisation des guillemets simples permet la résolution des variables contrairement au "single quote".

**Commande (s)**

>> var="Bonjour tout le monde"

>> echo $var

**Résultat (s)**

Bonjour tout le monde

**Commande (s)**

>> echo 'La valeur de var est $var'

**Résultat (s)**

La valeur de var est $var

### Backslash (\)

La barre oblique inversée (\) modifie la signification spéciale de l’apostrophe et des doubles guillemets, c'est-à-dire qu'elle permet d'échapper ou d'annuler la signification spéciale du caractère suivant.

**Commande (s)**

**>>** **fichier="/etc/resolv.conf"**

**>>** **echo "Le fichier est \"$fichier\" "**

**Résultat (s)**

Le fichier est "/etc/resolv.conf" …

**Commande (s)**

**>>** **fichier="/etc/resolv.conf"**

**>>** **echo "Le fichier est \$fichier"**

**Résultat (s)**

Le fichier est $fichier

|  |  |
| --- | --- |
| Caractère | Explication |
| **\a** | Alerte (cloche) |
| **\b** | Retour arrière (*backspace*) |
| **\n** | Nouvelle ligne |
| **\t** | Tabulation horizontale |
| **\v** | Tabulation verticale |
| **\\** | Barre oblique inversée |
| **\’** | Guillement simple |

**Exemples**

>> echo "Pizza Toute garnie \$15.50"

>> echo -e "\a Ding dong\a"

>> echo -e "Un chemin SMB doit être \\\\Nom\_du\_Serveur\\Nom\_du\_Partage"

>> echo -e "Nom :\t Savard "

>> echo -e "1\t Bombonne de gaz (20 litres)\t\$25,00"

>> echo -e "2\t Bombonne de gaz (5 litres)\t\$7,50"

L'**option -e** indique à la commande echo la **présence d’une séquence spéciale** (**\t** ou **\v** ou **\a** ou …).

**Autre exemple**

**Commande (s)**

>> echo "La nature, qui ne nous a donné qu'un seul organe pour la parole, \

nous en a donné deux pour l'ouïe, \

afin de nous apprendre qu'il faut plus écouter que parler. (Proverbe turc)"

**Résultat (s)**

La nature, qui ne nous a donné qu'un seul organe pour la parole, nous en a donné deux pour l'ouïe, afin de nous apprendre qu'il faut plus écouter que parler. (Proverbe turc)

### 

### Apostrophe inversés (*backquote*)

bash considère que les apostrophes délimitent une commande à exécuter.

**Commande (s)**

>> var=`date`

>> echo "La date et l’heuse sont … `date`"

**Résultat (s)**

La date et l’heure sont … mardi 13 février 2018, 22:30:17 (UTC+0000)

**Note** …

**`date`** et **$(date)** sont équivalents.

## Ligne de commandes

### Analyse de la ligne de commande

Lors de l’interprétation d’une commande, le shell commence par découper la ligne en mots séparés par des blancs.

Le premier mot attendu est le nom d'une commande.

Les mots suivants sont considérés comme des options ou des paramètres dont la compréhension incombe à la commande**.**

**Exemple**

En supposant que les comptes **usager1** à **usager20** sont déjà créés.

**Commande (s)**

>> grep -n usager /etc/passwd

La commande **grep** attend …

* des options précédées de - ;
* un modèle (expression rationnelle) des chaînes à chercher ;
* un ensemble de fichiers où elle doit chercher.

L’**option -n** de la précédente commande demande d’**afficher le numéro de ligne**.

### Enchaînement inconditionnel des commandes

En général, on place une commande par ligne que ce soit en ligne de commande ou dans un script.

Le **point virgule *;*** joue le rôle de **séparateur de séquence inconditionnel**.

Il permet ainsi d'écrire une séquence de plusieurs commandes sur une même ligne.

Toutes les commandes sont inconditionnellement exécutées (même si l'une d'entre elle provoque une erreur), et leurs résultats respectifs sont envoyés sur la sortie standard, séparés par un retour à la ligne "\n".

**Exemple**

**Commande (s)**

>> pwd ; ls -l

**Résultat (s)**

/home/lcsavard

total 8

-rw-r--r-- 1 root root 651 déc. 30 17:29 monsysteme.sh

drwxr-xr-x 2 root root 4096 déc. 31 15:27 scripts

**Commandes (s)**

>> clear ; ls -l

**Résultat (s)**

total 8

-rw-r--r-- 1 root root 651 déc. 30 17:29 monsysteme.sh

drwxr-xr-x 2 root root 4096 déc. 31 15:27 scripts

**Exemple**

La commande …

>> mkdir /usr/tux/{documents,textes,calculs,présentation}

est équivalente à …

>> mkdir /usr/tux/documents

>> mkdir /usr/tux/textes

>> mkdir /usr/tux/calculs

>> mkdir /usr/tux/présentations

**Exemple**

>> touch documents/fichier{01..09}

**Exemple**

La commande …

>> chown tux /home/kermit/{documents/{travaux,examens},lib/{fi?h?\*,how\_ex}}

est équivalente à …

>> chown tux /home/kermit/documents/travaux

>> chown tux /home/kermit/documents/examens

>> chown tux /home/kermit/lib/fi?h?\*

>> chown tux /home/kermit/lib/how\_ex

### Enchaînement conditionnel des commandes

Il est parfois nécessaire de savoir si une commande a réussie ou non avant d'en lancer une autre.

Les indicateurs **&&** et **||** permettent, respectivement, de lancer une commande si (et seulement si) la précédente a réussi ou échoué.

**&&** (ET) et **||** (OU) sur la ligne de commande sont des séparateurs qui jouent les rôles d'opérateurs **conditionnels**, en ce sens que la 2ème commande sera exécutée en fonction du code de retour de la 1ère commande.

**Avec la commande** …

commande1 && commande2

commande2 ne sera exécutée que si le code de retour de commande1 est 0 (exécution correcte).

**Avec la commande** …

commande1 || commande2

commande2 ne sera exécutée que si le code de retour de commande1 est différent de 0 (exécution erronée).

**Exemple**

La commande suivante …

**Commande (s)**

>> cd ~/tmp || mkdir $HOME/tmp

ne va exécuter la seconde commande que si la première échoue.

>> ls

**Résultat (s)**

calculs documents monsysteme.sh présentation scripts textes tmp

**Exemple**Si le fichier **document.texte existe**, le système retourne ...

>> **ls document.texte && echo "J'ai un fichier document.texte."**

**document.texte**

**J'ai un fichier document.texte**

Si le fichier document.pres n'existe pas, le système retournera un message d'erreur et le message de la commande echo ne s'affichera pas.

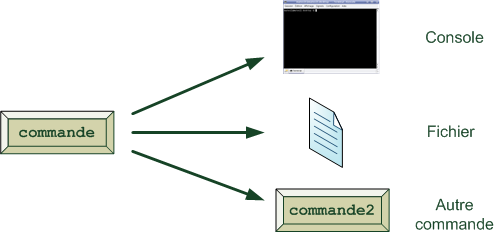
>> **ls document.texte && echo "J'ai un fichier documents.texte."**

**ls: impossible d'accéder à document.texte: Aucun fichier ou dossier de ce type**

### Redirections des entrées-sorties

Il est possible de rediriger le résultat d'une commande ailleurs que dans la console.

* **Où ?**   
  Dans un fichier   
  ou   
  En entrée d'une autre commande   
  afin de chaîner des commandes.   
  Ainsi, le résultat d'une commande peut en déclencher une autre.
* **Comment ?**   
  À l'aide de symboles spéciaux, appelés flux de redirection.



Toutes les commandes sont dotées par le système de **3 canaux de communication** …

**Commande (s)**

>> ls -l /dev/std\*

**Résultat (s)**

lrwxrwxrwx 1 root root 17 Aug 12 09:37 /dev/stderr -> ./proc/self/fd/2

lrwxrwxrwx 1 root root 17 Aug 12 09:37 /dev/stdin -> ./proc/self/fd/0

lrwxrwxrwx 1 root root 17 Aug 12 09:37 /dev/stdout -> ./proc/self/fd/1

|  |  |  |
| --- | --- | --- |
| **Périphérique** | **Descripteur de fichier** | **Description** |
| **/dev/stdin** | 0 | entrée standard (**stdin=standard input**) |
| **/dev/stdout** | 1 | sortie standard (**stdout=standard output**) |
| **/dev/stderr** | 2 | sortie des erreurs (**stdout=standard output**) |

**Exemple**

**Commande (s)**

>> ls > list

>> less list

**Résultat (s)  
calculs  
documents  
list  
monsysteme.sh  
présentation  
scripts  
textes  
tmp**

**Exemple**

**Commande (s)**

>> mkdir /root > log

**Résultat (s)**

mkdir: impossible de créer le répertoire « /root »: Le fichier existe

>> more log

---

**Redirection d’un résultat dans un fichier (> et >>)**

La redirection est une fonctionnalité de Linux qui permet de modifier les périphériques d'entrée/sortie standard lors de l'exécution d'une commande.

Le flux de travail de base de toute commande Linux est qu'elle prend une entrée et donne une sortie.

La manipulation la plus simple est d'écrire le résultat d'une commande dans un fichier, au lieu de l'afficher dans une console.

L’exercice suivant va utiliser le fichier **notes.csv** …  
**Tux,18/20,Excellent travail**

**Kermit,3/20,Nul comme d'hab**

**Grangallo,14/20,En nette progression**

**Squidly,9/20,Presque la moyenne**

**Gertrude,11/20,Pas mal mais peut mieux faire**

**Elmo,20/20,Toujours parfait**

**Bert,5/20,En chute libre**

La commande cut permet de « couper » une partie du fichier et d'afficher le résultat à la console.

**Commande (s)**

>> cut -d , -f 1 notes.csv

**Résultat (s)**

Tux

Kermit

Grangallo

Squidly

Gertrude

Elmo

Bert

Cette commande demande à cut de prendre tout ce qui se trouvait avant la première virgule afin d'avoir la liste des noms de tous les élèves présents …

* **d** indique le séparateur (*délimiter*) ;
* **f** indique le champ à découper (*field*) ;
* **notes.cvs** indique le fichier d’entrée.

Si on souhaite écrire le résultat de la commande (la liste des prénoms) dans un fichier, on utilise le chevron (>) qui permet de rediriger le résultat de la commande vers le fichier spécifié.

**sCommande (s)**

**>> cut -d , -f 1 notes.csv > eleves.texte**

**Note** …

On peut omettre certains espaces (**cut -d, -f1 notes.csv**).

Rien ne s'affiche dans la console car tout est redirigé dans le fichier nommé eleves.txt.   
Le fichier est créé s’il n’existe pas.   
Toutefois, s'il existe un fichier existant avec le même nom, la commande redirigée supprime le contenu de ce fichier pour le remplacer par le résultat de la commande.  
Les commandes **cat** ou **less** permet d’afficher le nouveau fichier créé à la console.

**Rappel** …  
Dans l’écosystème Linux, le **concept d'extension de fichier n’existe pas**.   
Le fichier aurait pu également se nommer **élèves**.

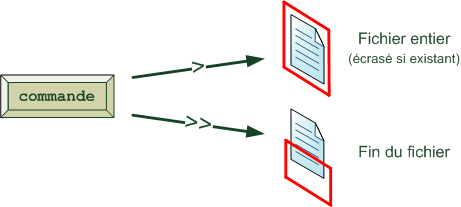
Le **double chevron (>>)** sert lui aussi à **rediriger le résultat d’une commande vers un fichier**, mais cette fois à la fin de ce dernier.

L’avantage du double chevron n'écrase pas le fichier s'il existe déjà.   
Si le fichier n'existe pas, il sera créé automatiquement.

**Commande (s)**

**>> cut -d, -f1 notes.csv >> eleves.texte**

… les noms seront ajoutés à la fin du fichier, sans écraser le résultat précédent.



**Redirection des erreurs**

**2> - 2>> - 2>&1**

Toutes les commandes produisent deux flux de données différents …

* la sortie standard  
  pour tous les messages (sauf les erreurs) ;
* la sortie d'erreurs  
  pour toutes les erreurs.

Par défaut, tout s'affiche dans la console : la sortie standard comme la sortie d'erreurs.

**Commande (s)**

**>> cut -d, -f1 fichier\_inexistant.csv > eleves.texte**

**Résultat (s)**

cut: fichier\_inexistant.csv: Aucun fichier ou répertoire de ce type

Puisque le fichier fichier\_inexistant.csv n'existe pas, l'erreur s'affiche dans la console au lieu d'avoir été envoyée vers le fichier eleves.texte.

On pourrait souhaiter enregistrer les erreurs dans un fichier à part pour ne pas les oublier et pour pouvoir les analyser ensuite.

Pour cela, on utilise l'opérateur 2>.

**Commande (s)**

**>> cut -d, -f1 fichier\_inexistant.csv > eleves.texte 2> erreurs.log**

Il y a deux redirections ici …

* **eleves.txt**  
  redirige le résultat de la commande (sauf les erreurs) dans le fichier eleves.txt.   
  C'est la sortie standard ;
* **2> erreurs.log**  
  redirige les erreurs éventuelles dans le fichier erreurs.log.   
  C'est la sortie d'erreurs.

**Fusion des sorties**

Il est possible de fusionner les sorties dans un seul et même fichier.

Il faut utiliser 2>&1.

Cet opérateur permet de rediriger toute la sortie d'erreurs dans la sortie standard.

**Commande (s)**

**>> cut -d, -f1 fichier\_inexistant.csv > eleves.texte 2>&1**

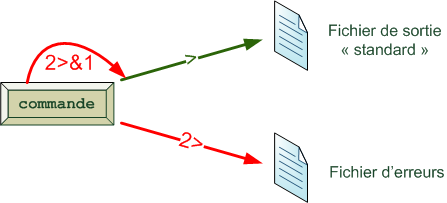
Le résultat (si cela a fonctionné), de même que les erreurs (s'il y a eu un problème) est redirigé vers le fichier eleves.texte.

**Remarque** …  
Il est possible d’utiliser l’opérateur 2>> pour rediriger les erreurs à la fin d'un fichier d'erreurs.

Toutefois, il n'est pas possible d’utiliser 2>>&1. En fait, le symbole 2>&1 envoie les erreurs dans le même fichier et de la même façon que la sortie standard.   
**Commande (s)**

**>> cut -d, -f1 fichier\_inexistant.csv >> eleves.texte 2>&1**

… les erreurs sont ajoutées à la fin du fichier **eleves.texte** comme le reste des messages.



On peut choisir de rediriger les erreurs dans un fichier à part (avec 2>) ou bien de les rediriger au même endroit que la sortie standard (avec 2>&1).

|  |  |
| --- | --- |
| **Fonction** | **Sortie** |
| **2>&1** | Redirige la sortie d'erreur (2) et la sortie standard (1) sur l'entrée de la commande suivante |
| ***fichier* 2>&1** | Redirige la sortie d'erreur et la sortie standard vers ***fichier*** |
| ***fichier* 2>> & 2>&1** | Redirige la sortie d'erreur et la sortie standard à la fin de ***fichier*** |

**Exemple**

**Commande (s)**

>> mkdir /root > log 2>&1

**Résultat (s)**

mkdir: impossible de créer le répertoire « /root »: Le fichier existe

Commande (s)

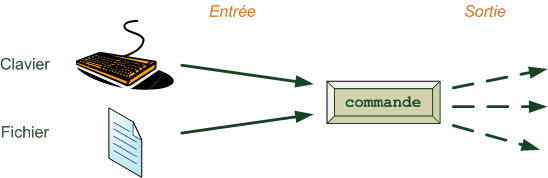
>> more log

**Résultat (s)**

mkdir: impossible de créer le répertoire « /root »: Le fichier existe

**Lecture depuis un fichier ou le clavier  
< - <<**

Habituellement, l'entrée provient des paramètres de la commande… mais on peut faire en sorte qu'elle vienne d'un fichier ou d'une saisie au clavier.



Le chevron ouvrant (<) permet d'indiquer d'où vient l'entrée qu'on envoie à la commande.

**Commande (s)**

**>> cat < resultats.csv**

Cela aura pour effet d'afficher le contenu du fichier envoyé en entrée …

**Résultat (s)**

**Tux,18/20,Excellent travail**

**Kermit,3/20,Nul comme d'hab  
…**

Saisir **cat < notes.csv** est strictement identique au fait d'écrire **cat notes.csv**… du moins en apparence.   
Le résultat produit est le même, mais ce qui se passe derrière est très différent.

* Si on saisit **cat notes.csv**, la commande cat reçoit en entrée le nom du fichier notes.csv qu'elle doit ensuite se charger d'ouvrir pour afficher son contenu.
* Si on saisit **cat < notes.csv**, la commande cat reçoit le contenu de notes.csv qu'elle se contente simplement d'afficher dans la console.   
  C'est le shell (le programme qui gère la console) qui se charge d'envoyer le contenu de notes.csv à la commande cat.

**Lecture depuis le clavier**

Le double chevron ouvrant (<<) permet d'envoyer un contenu à une commande avec le clavier.

**>> sort -n << FIN**

La console vous propose alors de saisir du texte …

**>**

La commande sort -n sert à trier des nombres.

On peut saisir une série de nombres, un par ligne (en appuyant sur la touche Entrée à chaque fois).

**> 13**

**> 132**

**> 10**

**> 131**

**>** **FIN**

La saisie de **FIN** termine la saisie.

Tout le texte écrit est alors envoyé à la commande (ici sort) qui traite cela en entrée.

**10**

**13**

**131**

**132**

**Exemple**

**Commande (s)**

**>> wc -m << FIN**

**>** **Combien de caractères dans cette phrase ?**

**>** **FIN**

**Résultat (s)**

**42**

**Remarque** …  
Dans les commandes précédentes, on peut remplacer FIN par une autre séquence de fin.

Rien n’oblige à écrire cette séquence en majuscules.

**Exemple**

**Commande(s)  
>> wc -m << STOP**

**>** **Combien de caractères dans cette phrase ?**

**>** **STOP**

**Résultat(s)  
42**

**Exemples**

**Commande (s)**

**>> sort -n << FIN > nombres.tri**

**> 13**

**> 132**

**> 10**

**> 131**

**>** **FIN**   
**Résultat (s)**

**10**

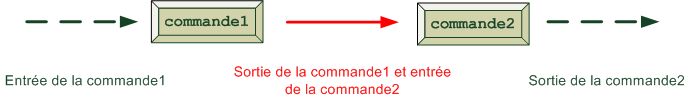
**13**

**131**

**132**

**Enchainement de commandes**

Il est possible de chaîner des commandes ou créer un tunnel signifie connecter la sortie d'une commande à l'entrée d'une autre commande.



Sommairement, le résultat de commande1 est immédiatement envoyé à la commande2.   
Il est possible de chaîner des commandes indéfiniment.

Cette fonctionnalité est vraiment une des plus importantes et décuple littéralement les possibilités offertes par la console. En pratique, chaque commande linux a un et un seul rôle. Toutefois, l'utilité de certaines commandes seules peut sembler limitée, mais celles-ci prennent en général tout leur sens lorsqu'on les combine à d'autres commandes.

**Exemple**

Avec cut, on peut récupérer les noms.

Avec sort, on peut les trier par ordre alphabétique.   
Il est possible de connecter le résultat de la commande cut à comme entrée à la commande sort pour avoir la liste des noms triés.

**>> cut -d, -f1 notes.csv | sort**

**Bert**

**Elmo**

**Gertrude**

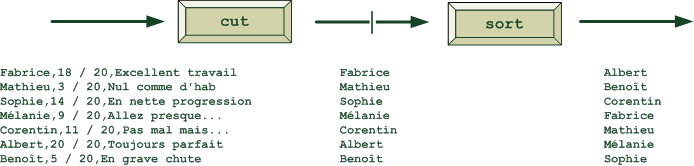
**Grangallo**

**Kermit**

**Squidly**

**Tux**

Le symbole pipe effectue la connexion entre la sortie de cut (des noms dans le désordre) et l'entrée de sort.



Il est aussi possible de rediriger le résultat dans un fichier.

**>> cut -d, -f1 notes.csv | sort > notes.tri**

**Résumé** …

Par défaut les canaux d'entrées et de sorties communiquent avec le clavier et l'écran.

Les commandes et les programmes qui ont besoin de données les attendent en provenance du clavier et expédient leurs résultats pour affichage sur le moniteur.

Il est possible de les détourner pour les rediriger vers des fichiers ou même vers les entrées-sorties d'autres commandes.

Pour ce faire, les symboles utilisés sont …

|  |  |
| --- | --- |
| Symbole | Description |
| > | Redirection de la sortie standard en direction d'un fichier  Le fichier est créé et écrase sans préavis le fichier existant portant le même nom |
| >> | Redirection de la sortie standard, à la fin du fichier s'il existe déjà |
| < | Redirection de l'entrée standard à partir d'un fichier |
| | | Enchaînement de commandes (appelé aussi tube ou *pipe*)  La sortie de la première commande est envoyée en entrée à la commande suivante |

**Exemple**

**Commande (s)**

>> ls --help | less

***Bit bucket***Lorsque l’on ne veut ni afficher le résultat d'une commande ou encore le stocker dans un fichier, il est possible de **rediriger le résultat vers** **/dev/null**.

C'est un peu le **« trou noir » de Linux** :   
tout ce qui est dirigé vers le *bit bucke*t disparaît immédiatement.

### Substitution de commande

La substitution de commande est un procédé permettant de substituer une commande par le résultat de son exécution. Le résultat est envoyé sur la sortie standard.

La commande doit être entourée de l'opérateur **` (***backquote*)ou être placée dans une parenthèse précédé de **$(...)**.

D'une manière générale, il est recommandé d'entourer l'expression de " "

**Exemples**

**Commande (s)**

>> echo "`whoami`, nous sommes le `date` "

>> echo "$(whoami), nous sommes le $(date) "

**Résultat (s)**

**tux, nous sommes le lundi 12 février 2018, 18:58:43 (UTC+0000)**

**tux, nous sommes le lundi 12 février 2018, 18:59:05 (UTC+0000)**

# Programmation bash

## Introduction

Un script bash est un fichier de type texte contenant une suite de commandes shell, exécutable par l'interpréteur (par exemple **/bin/bash**), comme une commande unique.   
Un script peut être lancé en ligne de commande, comme dans un autre script.

Toutefois, il s'agit bien plus qu'un simple enchaînement de commande : on peut définir des variables et utiliser des structures de contrôle, ce qui lui confère le statut de langage de programmation interprété et complet.

Le langage **Bash** gère notamment …

* les entrées-sorties et de leur redirection ;
* les variables (système, environnement et usager) ;
* le passage de paramètres ;
* les structures conditionnelles et itératives ;
* les fonctions internes.

## Création d’un script

Les lignes commençant par le caractère dièse # sont des commentaires.

Le script débute généralement par l'indication de son interpréteur écrite sur la première ligne : **#!/bin/bash**

**Exemple**Script dénommé **bonjour.sh**

#!/bin/bash

# script bonjour

# affiche un salut à l'utilisateur qui l'a lancé

# la variable d'environnement $USER contient le nom de login

echo ---- Bonjour $USER -----

# l'option -n empêche le passage à la ligne

# le ; sert de séparateur des commandes sur la ligne

echo -n "Nous sommes le " ; date

# recherche de $USER en début de ligne dans le fichier passwd

# puis extraction de l'uid au 3ème champ, et affichage

echo "Votre UID est " $(grep "^$USER" /etc/passwd | cut -d: -f3)

**Résultat (s)**

**---- Bonjour tux -----**

**Nous sommes le lundi 12 février 2018, 20:44:11 (UTC+0000)**

**Votre UID est 1000**

## Exécution du script

Il y a deux manières d’exécuter un script …

* En le précédent de la commande bash ;
* En attribuant au fichier (script) le droit **x** (soit exécutable) …

>> bash bonjour.sh

ou

>> chmod a+x bonjour

./bonjour

Dans cet exemple, .**/** indique le chemin du script, ici le répertoire courant.

Il est aussi possible d’indiquer le chemin absolu à partir de la racine (/). Ceci est le cas où le répertoire contenant le script n'est pas inclus dans le la variable d’environnement **PATH**.

Si les scripts personnels sont systématiquement stockés dans un répertoire précis, par exemple */home/bin*, on peut ajouter ce chemin dans le **PATH**.

Pour cela, il suffit d'ajouter un fichier contenant le code suivant dans **/etc/skel/**.

# bash\_profile

PATH=$PATH:$HOME/bin

Le répertoire **/etc/skel** contient les fichiers qui constitueront le *squelette* de tous les nouveaux comptes : tout ce qui est présent dans ce répertoire sera recopié par défaut dans le répertoire de chaque nouvel utilisateur.

Si une instruction incomplète est saisie en ligne de commande, l'interpréteur passe à la ligne suivante en affichant le prompt **>** et attend la suite de l'instruction.

Pour sortir, il suffit d’appuyer sur **Ctrl-C**.

## Mise au point (débogage)

Lorsque l'on programme en bash, il y a des options en ligne de commande afin de constater ce qui est lu et exécuté dans un script.

**Exécution en mode "trace" (-x) et en mode "verbose" (-v)**

**Commande (s)**

>> bash -x ./bonjour

Pour aider à la mise au point d'un script, on peut insérer des lignes temporaires …

* **echo $var** pour afficher la valeur de la variable ;
* **exit 1**  pour forcer l'arrêt du script à cet endroit.

On peut passer des arguments à la suite du nom du script, séparés par des espaces.

Les valeurs de ces paramètres sont récupérables dans le script grâce aux paramètres de position $1, $2 .. mais, contrairement aux langages de programmation classiques, ils ne peuvent pas être modifiés.

**Exemple**

**Commande (s)**

>> nano sortie.sh

#!/bin/bash

# appel du script : ./bonjour nom prénom

if [ $# -eq 2 ]

then

echo "Bonjour $1 $2 et bonne journée !"

else

echo "Syntaxe : $0 nom prénom"

fi

**Commande (s)**

>> bash sortie.sh Louis Savard

**Résultat (s)**

**Bonjour Savard Louis et bonne journée !**

**Commande (s)**

>> bash sortie.sh

**Résultat (s)**

Syntaxe : sortie nom prénom

## 

## Entrées-Sorties

Les entrées et sorties sont les voies de communication entre le programme bash et la console …

**Commande echo**La commande **echo** affiche un argument texte entre guillemets sur la sortie standard.

La validation d'une commande **echo** provoque un saut de ligne.

**Commande (s)**

>> echo "Bonjour à tous !"

**Résultat (s)**

**Bonjour à tous !**

Il est possible d’insérer les caractères spéciaux habituels, caractères qui seront interprétés seulement si l'option **-e** suit **echo**.

* \n (saut ligne) ;
* \b retour arrière) ;
* \t (tabulation) ;
* \a (alarme) ;
* \c (fin sans saut de ligne).

**Commande (s)**

>> echo "Bonjour \nà tous !"

**Résultat (s)**

**Bonjour \nà tous !**

**Commande (s)**

>> echo -e "Bonjour \nà tous !"

**Résultat (s)**

**Bonjour**

**à tous !**

**Commande (s)**

>> echo -e "Bonjour \nà toutes \net à tous !"

**Résultat (s)**

**Bonjour**

**à toutes**

**et à tous !**

**Commande read**

**read** permet l'affectation directe par la lecture de la valeur saisie sur l'entrée standard au clavier apr un usager.

***read var1 var2*** ... attend la saisie au clavier d'une liste de valeurs pour les affecter, après la validation globale, respectivement aux variables *var1*, *var2*, …

**Commande (s)**

>> echo "Donnez votre prénom et votre nom"

>> read prenom nom

>> echo "Bonjour $prenom $nom"

**Résultat (s)**

**Bonjour Louis Savard**

## 

## Les variables bash

### Variables utilisateur

Comme dans tous les langages de programmation, on trouve en bash ce que l'on appelle des **variables**. Les variables permettent de stocker temporairement des informations en mémoire. C'est en fait la base de totu script.

Syntaxe

variable=valeur

Le signe = **NE DOIT PAS** être entouré d'espace(s)

On peut initialiser une variable à une chaîne vide …

>> chaine\_vide=

Si valeur est une chaîne avec des espaces ou des caractères spéciaux, il faudra alors l'entourer de   
guillements simples (**' '**) ou de guillemets doubles (**" "**).

Le caractère **\** permet de masquer le sens d'un caractère spécial comme **"** ou **'.**

Lors de l’utilisation d’une variable, il fait faire précéder son nom du signe **$** afin de faire référence à sa valeur.

Pour afficher toutes les variables …

**Commande (s)**

>> set

**Résultat (s)**

**BASH=/bin/bash**

**…**

**HOME=/root**

**HOSTNAME=srv04.profsavard,info**

**…**

**PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin**

**…**

**PS1='${debian\_chroot:+($debian\_chroot)}\u@\h:\w\$ '**

**…**

**USER=root**

**…**

Afin d’empêcher la modification d'une variable, il faut invoquer la commande **readonly.**

>> readonly pi=3.14

### Assignation d’une valeur à une variable

L’assignation d’une valaeur à une variable se fait de la manière suivante …

**Commande (s)**

>> total=23

>> nom=Tux

>> nom="Tux Tremblay"

### Substitution de variable

Si une chaîne contient la référence à une variable, le shell doit d'abord remplacer cette référence par sa valeur avant d'interpréter la phrase globalement.

**Exemple**

**Commande (s)**

>> n=abc

>> echo "La variable \$n vaut $n"

**Résultat (s)**

**La variable $n vaut abc**

**Commande (s)**

>> salut="Bienvenue à tous !"

>> echo "Message1 : $salut"

**Résultat (s)**

**Message1 : Bienvenue à tous !**

**Commande (s)**

>> echo 'Message2 : $salut'

**Résultat (s)**

**Message2 : $salut**

**Commande (s)**

>> echo "Message3 : \"$salut\" "

**Résultat (s)**

**Message3 : "Bienvenue à tous !"**

**Commande (s)**

>> readonly salut

**>> salut="simple bonjour"**

**Résultat (s)**

**bash: salut : variable en lecture seule**

**Commande (s)**

**>> echo "Message4 : $salut"**

**Résultat (s)**

**Message4 : Bienvenue à tous !**

### Variables exportées

Toute variable est définie dans un shell spécifique.

Afin qu’elle devienne globale, elle doit être exportée par la commande **export** …

>> export variable

>> export variable=valeur

>> export --> Pour obtenir la liste des variables exportées

**Exemple**

Commande (s)

>> export institution=Maisonneuve

### Opérateur {}

Dans certains cas en programmation, on peut être amené à utiliser des noms de variables incluses dans d'autres variables.

Comme il n'y a pas de substitution automatique, la présence d’accolades {} force l'interprétation des variables incluses.

**Exemples**

**Commande (s)**

>> rep\_user="/home/public"

>> echo $rep\_user

**Résultat (s)**

**/home/public**

**Commande (s)**

>> u1=$rep\_user1

>> echo "cas1: u1="$u1

**Résultat (s)**

**cas1: u1=**

**Commande (s)**

>> u1=${rep\_user}1

>> echo "cas2: u1="$u1

**Résultat (s)**

**cas2: u1=/home/public1**

### Variables d'environnement

Les variables d'environnement constituent un moyen d'influencer le comportement certains logiciels sur un système donné. Par exemple, la variable d'environnement **LANG** détermine la langue que les logiciels utilisent afin de communiquer avec l'utilisateur

Cette liste en est accessible par la commande …

>> env

Les plus utiles sont $HOME,**$PATH**,**$USER**,**$PS1**,**$SHELL**,**$ENV**,**$PWD**, …

La commande **echo** permet d'obtenir la valeur d'une telle variable.

Par exemple …

**Commande (s)**

>> echo $PATH

>> echo $USER

**Résultat (s)**

**/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games**

**lcsavard**

Pour modifier le chemin existant (*PATH*), il faut le modifier avec la commande suivante …

**Commande (s)**

>> PATH=$PATH:/home/lcsavard/bin

Pour ajouter le répertoire courant (non présent par défaut) …

**Commande (s)**

>> PATH=$PATH:./

Attention à ne pas écraser la liste des chemins existants (**PATH** en majuscules !)

La variable **$HOME** contient le chemin du répertoire personnel.   
La commande **cd $HOME** peut être abrégée en **cd**.

La variable **$USER** contient le nom de l'usager.

**Rappel …**

Une variable de shell peut être exportée afin devenir une variable d'environnement grâce à la commande **export**.

Pour créer la variable d'environnement **EDITOR** et lui assigner la valeur **nano**, plusieurs méthodes peuvent être utilisée.

**Commande (s)**

>> EDITOR=nano

>> export EDITOR

>> env | grep –I editor

**Résultat (s)**

**EDITOR=nano**

**Commande (s)**

>> export EDITOR=nano

**Résultat (s)**

**EDITOR=nano**

La variable **$SHLVL** donne le niveau du *shell* courant.

**Exemple d’utilisation de variables d’environnement**

**Commande (s)**

>> nano var\_env.sh

nom=Louis Savard

message="Je m'appelle $nom"

echo Aujourd\'hui, quel jour sommes-nous? ; read jour

echo Aujourd\'hui $jour, $message, usager:$USER, station:$HOSTNAME

>> bash var\_env.sh

**Résultat (s)**

**Aujourd'hui, quel jour sommes-nous?**

**Jeudi**

**Aujourd'hui Jeudi, Je m'appelle lcsavard, usager:lcsavard, station:srv01.profsavard.info**

### Variables système

Les variables système sont gérées par le système et s'avèrent très utiles dans les scripts.   
Elles ne sont accessibles qu'en lecture.

Ces variables sont automatiquement affectées lors d'un appel de script suivi d'une liste de paramètres.

Leurs valeurs sont récupérables dans **$1**, **$2** ...**$9**.

|  |  |
| --- | --- |
| **Variable** | **Résultat** |
| **$?** | Valeur de sortie de la dernière commande  Vaut 0 si la commande s'est déroulée sans problème |
| **$0** | Variable qui contient le nom du script |
| **$1 à $9** | Paramètres passés à l'appel du script |
| **$#** | Nombre de paramètres passés au script |
| **$\*** | Liste des paramètres à partir de $1 |
| **$$** | N° (PID) du processus courant |
| **$!** | N° (PID) du processus fils |

**Exemples**

if [ $# -eq 0 ]

then

echo "pas de nom"

else

echo "Votre nom est "$1

fi

**Commande (s)**

>> echo "nombre de paramètres :"$#

>> echo "nom du programme :"$0

>> echo "liste des paramètres :"$\*

**Résultat (s)**

**nombre de paramètres :0**

**nom du programme :-bash**

**liste des paramètres :**

### 

### Paramètres de position

On peut récupérer facilement les compléments de commande passés sous forme d'arguments sur la ligne de commande, à la suite du nom du script, et les utiliser pour effectuer des traitements.

Ce sont les variables système spéciales **$1** , **$2** .. **$9** appelées paramètres de position.

Celles-ci prennent au moment de l'appel du script, les valeurs des chaînes passées à la suite du nom du script.

* le nombre d'argument est connu avec **$#** ;
* la liste complète des valeurs des paramètres s'obtient avec **$\***.

## Commandes importantes

### shift

**shift** effectue un décalage de pas **+1** dans les variable **$**.

Ainsi **$1** prend la valeur de **$2** et ainsi de suite.

**Exemple**

**Commande (s)**

>> a=1 ; b=2 ; c=3

>> set $a $b $c

>> echo $1, $2, $3

**Résultat (s)**

**1, 2, 3**

**Commande (s)**

>> shift

>> echo $1, $2, $3

**Résultat (s)**

**2, 3,**

Tr  
Cette commande de filtre permet d'effectuer des remplacements de caractères dans une chaîne.

**Transformer une chaîne en minuscules**

Commande (s)

>> chaine="BONJOUR"

>> echo $chaine | tr 'A-Z' 'a-z'

**Résultat (s)**

**bonjour**

Afin de permettre l'utilisation de la commande **set**, il est nécessaire que le séparateur de champ sur une ligne soit l'espace comme dans l’exemple suivant.

Pour créer un fichier **passwd.txt** qui introduit un espace à la place de ":" á partirde **/etc/passwd** …

**Commande (s)**

>> cat /etc/passwd | tr ":" " " | grep lcsavard > passwd.txt

>> cat passwd.txt

**lcsavard x 1000 1000 Louis C. Savard,,, /home/lcsavard /bin/bash**

**Résultat (s)**

**lcsavard x 1000 1000 Louis C. Savard,,, /home/lcsavard /bin/bash**

### set

Cette commande interne est très pratique afin de séparer une ligne en une liste de mots, chacun de ces mots étant affecté à une variable positionnelle.

Comme vu précédemment, le caractère de séparation est l'espace.

**Exemples**

**Commande (s)**

**>> nano couleurs.sh**

#!/bin/bash

# Soit une chaîne couleurs qui contient une liste de mots

couleurs="rouge vert bleu noir"

# La commande set va lire chaque mot de la liste et l'affecter aux paramètres

set $couleurs

echo $1 $2 $3

shift

echo $1 $2 $3

**Résultat (s)**

**rouge vert bleu**

**vert bleu noir**

**Commande (s)**

>> nano valeurs.sh

#!/bin/bash

a=1 ; b=2 ; c=3

set $a $b $c

echo $1, $2, $3

**Résultat (s)**

**Les valeurs de a, b, c sont récupérées dans $1, $2, $3**

### eval

Cette commande ordonne l'interprétation par le hell de la chaîne passée en argument.

On peut ainsi construire une chaîne que l'appel à **eval** permettra d'exécuter comme une commande …

**Exemple**

>> nano message02.sh

message="Quelle est la date d'aujourd'hui"

set $message

echo $# ---> le nombre de mots est 5

echo $1

echo $4 ---> affiche la chaîne "date"

echo $5

eval $4 ---> interprète la chaîne "date" comme une commande

**Résultat (s)**

**5**

**Quelle**

**date**

**d’aujourd’hui**

**jeudi 15 février 2018, 15:56:34 (UTC+0000)**

**eval** est particulièrement pratique afin de construire une chaîne dont la valeur sera égale au libellé d'un enchaînement de commandes.

Pour faire exécuter ces commandes contenues dans la chaîne, on la passe comme argument de la commande **eval …**

**Commande (s)**

>> liste="date;who;pwd"

>> eval $liste ---> exécute bien les 3 commandes

**Résultat (s)**

**jeudi 15 février 2018, 16:07:24 (UTC+0000)**

**lcsavard pts/0 2018-02-15 13:36 (206.167.44.26)**

**/home/lcsavard**

Remarque …

Dans cet exemple, le ; est un séparateur de commandes.

Soit la chaîne **$user** qui contient des informations sur un compte à créer.

S'il utilise un autre séparateur que ";" on fait appel à **tr** en premier …

**Commande (s)**

>> nano creation\_usager.sh

#!/bin/bash

addgroup comptabilite

user="login=usager8 ; motdepasse=h34345 ; nom=Jean ; groupe=comptabilite"

eval $user

echo $login $motdepasse $nom $groupe

useradd -G $groupe $login

echo $login:$motdepasse | chpasswd

>> cat /etc/passwd | grep usager8

**Résultat (s)**

**usager8:x:1001:1002::/home/usager8:**

### basename

basename permet d’éliminer le chemin d’accès et le suffixe d’un nom de fichier.

Syntaxe : basename nom [suffixe]

**Exemples**

**Commande (s)**

>> basename /etc/lpd.conf .conf *-->* lpd

Si un suffixe est indiqué, et s’il est identique à la partie finale du nom, il est éliminé de celui-ci.

>> basename /etc/named.conf .conf *--> named*

### seq

seq permet de générer une séquence de nombres.

Syntaxes …

* seq [options] dernière valeur
* seq [options] première et dernière valeur
* seq [options] première valeur increment et dernière valeur

**Exemples**

**Commande (s)**

**>> seq 30**

**Résultat (s)**

**1**

**2**

**3**

**…**

**28**

**29**

**30**

**Commande (s)**

>> seq 10 30

Résultat (s)

**10**

**11**

**12**

**…**

**28**

**29**

**30**

**Commande (s)**

>> seq -s : 30

**Résultat (s)**

**1:2:3:4:5:6:7:8:9:10:11:12:13:14:15:16:17:18:19:20:21:22:23:24:25:26:27:28:29:30**

**Commande (s)**

>> seq 1 2 40

**Résultat (s)**

**1**

**3**

**5**

**7**

**…**

**37**

**39**

### test

Comme son nom l'indique, la commande **test** sert à vérifier des conditions.

Ces conditions peuvent porter sur des fichiers, des chaînes ou une expression numérique.

Cette commande courante sert donc à prendre des décisions, d'où son utilisation comme condition dans les structures conditionnelles **if.. then ..else**.

Elle peut remplacer, en quelque sorte, des variables booléennes... qui n'existent pas.

La commande **test** admet 2 syntaxes (la seconde étant la plus utilisée) …

**# test expression**

ou

**# [ expression ]**

**Attention aux espaces autour de … expression.**

**Valeur de retour**

* Chaque commande transmet au programme appelant un code, appelée valeur de retour (*exit status*) qui stipule la manière dont son exécution s'est déroulée.   
  Cette valeur numérique est stockée dans la variable spéciale **$?**
* Par convention avec bash, **la valeur de retour est toujours 0 si la commande s'est déroulée correctement**, sans erreur.
* Une valeur de retour différente de 0 signale donc une erreur, qui peut être éventuellement analysée selon cette valeur.
* Une variable système spéciale **$?** contient la valeur de retour de la précédente commande.   
  On peut afficher cette valeur avec la commande echo.

**Exemples**

**Commande (s)**

>> test 100 -gt 99 && echo "Oui, cela est vrai." || echo "Non, cela est faux."

**Résultat (s)**

**Oui, cela est vrai.**

**Commande (s)**

>> test 100 -lt 99 && echo "Oui." || echo "Non."

**Résultat (s)**

**Non.**

Commande (s)

>> [ "génial" = "génial" ]; echo $?

**Résultat (s)**

**0**

**Commande (s)**

>> [ 5 -eq 6 ]; echo $?

**Résultat (s)**

**1**

**Autres exemples**

**Commande (s)**

>> ls ~

>> echo $?

**Résultat (s)**

**0**

**Commande (s)**

>> ls /abcd

>> echo $?

**Résultat (s)**

**1 ----> si abcd n'existe pas**

**Tester un fichier**

# [ option chaine ]

Tableau des principales options

|  |  |
| --- | --- |
| **Option** | **Signification** |
| -e | Le fichier existe (oui ou non) ? |
| -f | Un fichier normal (oui ou non) ? |
| -d | Un répertoire (oui ou non) ? |
| -r | Le fichier est lisible (oui ou non) ? |
| -w | Le fichier est modifiable (oui ou non) ? |
| -x | Le fichier est exécutable (oui ou non) ? |
| -s | Le fichier n'est pas vide (oui ou non) ? |

**Exemples**

|  |  |
| --- | --- |
| **Commande** | **Résultat** |
| [ -s $1 ] | **Vrai (renvoie 0) - si le fichier passé en argument n'est pas vide** |
| [ $# -eq 0 ] | **Le nombre d'arguments est 0** |
| [ -w fichier ] | **Le fichier est-il modifiable ?** |
| [ -r "/etc/passwd" ] | **Le fichier/etc/passwd est-il lisible ? echo $? --> 0 (vrai)** |
| [ -r "/etc/shadow" ] | **Le fichier/etc/shadow est-il lisible ? echo $? --> 1 (faux)** |
| [ -r "/etc/shadow" ] ||  echo "échec de lecture" | **Le fichier/etc/shadow est-il lisible ?** |

**Tester une chaîne**

# [ option chaine ]

Tableau des principales options

|  |  |
| --- | --- |
| **Option** | **Signification** |
| -z | La chaine est-elle vide (oui ou non) ? |
| -n | La chaine n’est-elle pas vide (oui ou non) ? |
| = | Les chaînes comparées sont-elles identiques (oui ou non) ? |
| != | Les chaînes comparées sont-elles différentes (oui ou non) ? |

**Exemples**

|  |  |
| --- | --- |
| **Commande** | **Résultat** |
| [ -n "non vide" ] | **La chaine est-elle vide (oui ou non) ?  echo $? --> 0 (vrai)** |
| chaine="bonjour" ; [ $chaine = "bonjour" ] | **Les chaines sont-elles identiques ? echo $? --> 0 (vrai)** |
| [ $USER != "root" ] &&  echo "l'utilisateur n'est pas root" | **L’usager connecté est-il root ? echo $? --> 0 (vrai)** |

**Tester un nombre**

# [ nb1 option nb2 ]

Note … Il y a une conversion automatique de la chaîne de caractères en nombre.

|  |  |  |
| --- | --- | --- |
| **Opérateur** |  | **Signification** |
| -eq | = | Égal |
| -ne |  | Différent |
| -lt | < | Strictement inférieur |
| -gt | > | Strictement supérieur |
| -le | <= | Inférieur ou égal |
| -ge | >= | Supérieur ou égal |

**Exemples**

|  |  |
| --- | --- |
| **Commande** | **Résultat** |
| # [ 2 = 2 ]  # echo $? | **0** |
| $ [ 2 = 3 ]  $ echo $? | **1** |
| # a=2 ; [ $a -lt 9 ] ; echo $? | **0 ----> 2 est stritement inférieur à 9 - l’expression est vraie** |

**Opérations booléennes**

Les opérations booléennes sont des opérations logiques dans lesquelles les opérandes[[1]](#footnote-1) et les résultats ne prennent que deux valeurs, soit vrai ou faux.

On utilise généralement les symboles 0 et 1 à la place de vrai et faux.

|  |  |
| --- | --- |
| **Option** | **Signification** |
| [ expression1 -a expression2 ] | (**and**) 0 si les 2 expressions sont vraies |
| [ expression1 -o expression2 ] | (**or**) 0 si l'une des 2 expressions est vraie |
| [ ! expression1 ] | Négation |

**Exemples**

**Commande (s)**

>> f="/root" ; [ -d $f -a -x $f ] ; echo $?

**Résultat (s)**

**0**

**Commande (s)**

>> val=80; [ $val -lt 20 -o $val -ge 41 ] || echo "val entre 20 et 40"

Résultat (s)

**---**

**Opérations mathématiques**

Si dans les scripts bash il faut effectuer des opérations mathématique il existe une méthode très simple.

|  |  |  |
| --- | --- | --- |
| **Opération** | **Commande** | **Résultat** |
| Addition | >> echo $((7+4)) | 11 |
| Soustraction | >> echo $((8-2)) | 6 |
| Multiplication | >> echo $((4\*7)) | 28 |
| Division | >> echo $((8/2)) | 4 |

Il est aussi possible d'effectuer ces opérations avec des variables comme par exmple …

|  |  |  |
| --- | --- | --- |
| **Opération** | **Commande** | **Résultat** |
| Division | >> i=4  >> echo $(($i/2)) | 2 |

Il est aussi possible d’utiliser les opérateurs logiques.

**let** permet d’effectuer une incrémentation …

|  |  |  |
| --- | --- | --- |
| **Opération** | **Commande** | **Résultat** |
| Incrémentation | >> i=4  >> let i++  >> echo $i | 5 |
| Décrémentation | >> i=5 >> let i--  >> echo $i | 4 |

[**https://ryanstutorials.net/bash-scripting-tutorial/**](https://ryanstutorials.net/bash-scripting-tutorial/)

1. Nom parfois donné à chacune des données qui interviennent dans une opération.   
   Dans l’opération d’addition 12 + 45 = 57, les opérandes sont les termes 12 et 45.  
   Dans l’opération de multiplication 12 × 7 = 84, les opérandes sont les facteurs 12 et 7. [↑](#footnote-ref-1)