Shell + Bash Scripting

Par Cyril VIMARD

Plan du cours

- Exécution bash
  - Méthode Graphique
  - Méthode dans un terminal
- Les différents types de shells
- Les variables
- Shebang
- Les arguments en ligne de commande
- L'arithmétique
- La commande test
- Les crochets
- Les opérateurs logiques
- Table de vérité de “-a”
- Table de vérité de “-o”
- La structure : “if”
- Les structures while et until
- La structure case
- Evaluation

Les prés requis

A faire obligatoirement !

Script BASH

- Vous devez disposez d'un des systèmes d'exploitation suivant installer sur votre machine : **OSX/Unix** ou **Ubuntu**
- Vous devez via le terminal utiliser ou installer les commandes suivantes sur votre système d'exploitation via **ubuntu/linux** la commande **apt** ou **homebrew** sur OSX.
- Suite à la mise à jour de votre système via la commande apt ou homebrew vous devez installer les packages suivants :
  - **zsh**
  - **curl**
  - **git**
  - **vim**
  - **htop**
  - **oh-my-zsh**
- Avec zsh vous devrez faire en sorte que (la dernière version):
  - zsh --version
  - zsh *.* (x86_64-ubuntu-linux-gnu)
- Avec **oh-my-zsh** vous devrez faire en sorte que le thème soit configuré avec **"amuse"**. Avec **git** vous devrez faire en sorte que (la dernière version) :
  - git --version
  - git version 2.25.1

Le Scripts BASH

Les notions

Script BASH

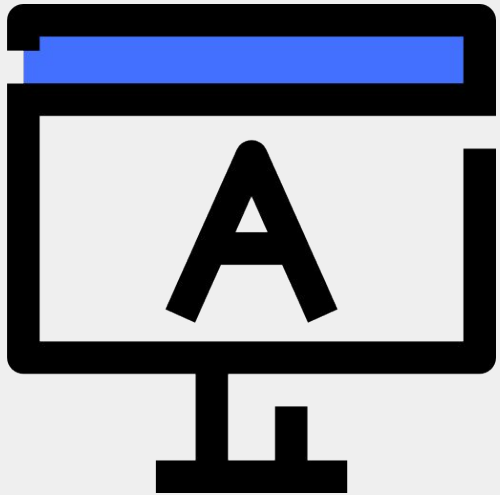
Un script shell permet d'automatiser une série d'opérations. Il se présente sous la forme d'un fichier contenant une ou plusieurs commandes qui seront exécutées de manière séquentielle.

Pour faire qu'un script soit exécutable avec la méthode **graphique** ou du **Terminal.**

Doc en FR

Méthode Graphique

Exécuter un script bash



Script BASH

Votre script est un simple fichier texte, par défaut il s'ouvre donc avec l'éditeur de texte défini par défaut (ex : Gedit dans une session Unity ou Gnome). Pour qu'il soit autorisé à se lancer en tant que programme, il faut modifier ses propriétés. Pour cela faites un clic droit sur son icône, et dans l'onglet "Permissions" des "Propriétés", cocher la case "autoriser l'exécution du fichier comme un programme".

Par la suite, un double-clic sur l'icône vous laissera le choix entre afficher le fichier (dans un éditeur de texte) et le lancer (directement ou dans un terminal pour voir d'éventuels messages d'erreurs)

Par ailleurs Nautilus ne propose pas de lancer le script par simple clic avec les réglages de bases. Il faut aller dans **Menu→ Édition → Préférences → Onglet comportement → fichier texte et exécutable et cocher pour fichiers exécutables Demander à chaque fois**.

Sous Lubuntu, si cette méthode ne fonctionne pas, vous devez d'abord effectuer l'opération suivante :

- Dans le menu principal, allez sur Outils système et faites un **clic droit → Propriétés sur le raccourci vers le terminal**. Notez le contenu du champ Commande et annulez.
- Ouvrez votre gestionnaire de fichier PCManFM et allez dans le menu supérieur sur **éditer → Préférences puis dans la fenêtre qui s'ouvre sélectionnez Avanc**é.
- Remplacez le contenu du champ Terminal emulator par le contenu du champ Commande que vous avez pris soin de noter à la première étape.
- Vous pouvez ensuite suivre la méthode graphique indiquée ci-dessus pour exécuter vos scripts shell.

Méthode dans un Terminal

Nous utiliserons cette approche !

Script BASH

Il suffit de se placer dans le dossier où est le script, et de lancer :

```
bash nom_du_script
```

Mais pas toujours bash (dépend du langage du script) ou si vous voulez l'exécuter par son nom , il faut le rendre exécutable avec chmod. Pour ceci tapez la commande qui suit :

```
chmod +x nom_du_script
```

Puis vous pouvez exécuter le script en faisant :

```
./nom_du_script
```

Mais pourquoi le **./** ?, Il peut être intéressant d'ajouter un répertoire au "**PATH**" pour pouvoir exécuter ses scripts sans avoir à se placer dans le bon dossier. Je m'explique, quand vous tapez une commande ("**ls**" par exemple), le shell regarde dans le **PATH** qui lui indique où chercher le code de la commande.

Pour voir à quoi ressemble votre **PATH**, tapez dans votre console:

```
echo $PATH
```

Cette commande chez moi donnait initialement :

```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

C'est à dire que le shell va aller voir si la définition de la commande tapée ("ls" pour continuer sur le même exemple) se trouve dans **/usr/local/bin** puis dans **/usr/bin...** jusqu'à ce qu'il la trouve.

Script BASH

Ajouter un répertoire au PATH peut donc être très pratique. Par convention, ce répertoire s'appelle bin et se place dans votre répertoire personnel. Si votre répertoire personnel est **/home/toto**, ce répertoire sera donc **/home/toto/bin**. Pour pouvoir utiliser mes scripts en tapant directement leur nom (sans le "./") depuis n'importe quel répertoire de mon ordinateur, il me suffit d'indiquer au shell de chercher aussi dans ce nouveau dossier en l'ajoutant au **PATH**. Pour ceci, il suffit de faire :

```
export PATH=$PATH:$HOME/bin
```

La commande

```
echo $PATH
```

retourne maintenant

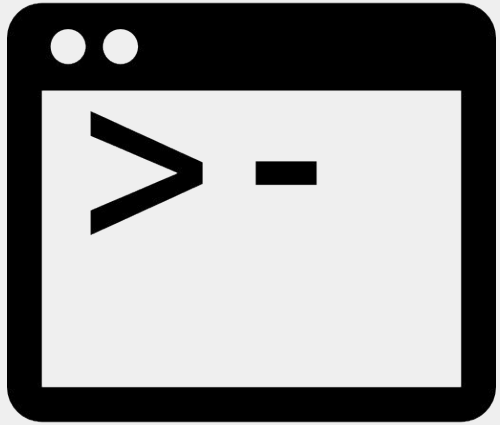
```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/home/toto/bin
```

et je peux lancer le script appelé "monScript" situé dans **"/home/toto/bin"** en tapant directement : **monScript**

*Cette procédure est pour une modification temporaire du **PATH** et qui sera donc effacée à la fin de la session. Pour rendre la modification permanente, ajouter la commande dans le fichier texte caché **.bashrc** se trouvant dans votre dossier personnel ainsi que dans le dossier **/root**.*

*Dans les dernières versions de ubuntu (12.04 +) si le dossier **$HOME/bin** existe il est automatiquement ajouté au **PATH**. La commande est incluse dans le fichier **~/.profile** lancé lors de toutes sessions (graphique ou console).*

Les **différents types de shells**



Les différents types de shells

Comme vous avez sûrement dû l'entendre, il existe différents types de shells ou en bon français, interpréteurs de commandes :

- **dash** (Debian Almquist shell) : shell plus léger que bash, installé par défaut sur Ubuntu ;
- **bash** (Bourne Again SHell) : conçu par le projet GNU, shell linux ; le shell par défaut sur Ubuntu ;
- **rbash** : un shell restreint basé sur bash. Il existe de nombreuses variantes de bash ;
- **csh**, **tcsh** : shells C, créés par Bill Joy de Berkeley ;
- **zsh**, **shell C** écrit par Paul Falstad ;
- **ksh** (⇔ ksh88 sur Solaris et équivaut à ksh93 sur les autres UNIX/Linux cf.Korn shell History): shells **korn** écrits par David Korn, **pdksh** (Public Domain Korn Shell ⇔ ksh88) ;
- **rc** : shell C, lui aussi conçu par le projet GNU ;
- **tclsh** : shell utilisant Tcl ;
- **wish** : shell utilisant Tk .

Il existe bien entendu beaucoup d'autres types de shells. Pour savoir quel type de shell est présent sur une machine, aller dans un terminal et taper la commande **ps**.

La commande **sh** est en fait un lien symbolique vers l'interpréteur de commandes par défaut : **/bin/dash.**

#! Shebang

Les différents types de shells

Le shebang, représenté par **#!**, est un en-tête d'un fichier texte qui indique au système d'exploitation (de type Unix) que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

Son nom est possiblement un mot-valise pour sharp (dièse, désignant ici improprement le croisillon, #) et bang (désignant ici le point d'exclamation, !).

On trouve aussi d'autres dénominations : **sha-bang, shabang, she-bang, hash-bang.**

Tout de suite après le shebang se trouve un chemin d'accès (exemple : #!/bin/sh). Il est possible d'ajouter une espace entre le point d'exclamation et le début du chemin d'accès.

Ainsi la séquence **#! /bin/bash** est valide et équivalente à **#!/bin/bash**. Le chemin d'accès est le chemin vers le programme qui interprète les commandes de ce script, qu'il soit un shell, un langage de script, un langage de programmation ou un utilitaire. On peut le faire suivre des options qu'il reconnaît (**par exemple -x pour un shell pour afficher le détail de son exécution**). Ensuite, cet interpréteur de commandes exécute les commandes du script, en commençant au début (ligne 1), en ignorant les commentaires.

```
#Exemples de shebang :

#!/bin/sh -x
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
#!/usr/bin/python -O
```

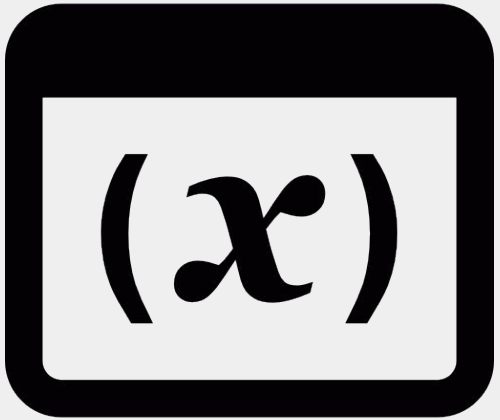
*On peut utiliser la commande env au lieu d'un interpréteur de commandes pour chercher celui-ci dans le **PATH** (ce qui évite de devoir réécrire la première ligne des scripts si on doit les porter sur une autre machine par exemple)*

#!/usr/bin/env python

… mais le problème est que l'on risque de ne pas appeler la bonne version de l'interpréteur si plusieurs sont présentes sur la machine. Un moyen de pallier cela est de détailler le nom de l'interpréteur :

#!/usr/bin/env python2

Les **variables**



Les variables

Il faut savoir que en bash les variables sont toutes des chaînes de caractères. Cela dépendra de son USAGE, pour une opération arithmétique prochaine voir : let ma_variable sinon pour conserver une valeur : il suffit de lui donner un nom et une valeur avec l'affectation égale :

```
ma_variable=unmot
```

Ici la valeur est affectée à la variable ma_variable .Attention: pas d'espace ni avant ni après le signe "=" . Autre exemple avec une commande avec arguments :

```
nbre_lignes=$(wc -l < fichier.ext)
```

nbre_lignes contiendra le nombre de lignes contenu dans fichier.ext . Pour voir le contenu d'une variable, on utilisera echo (par exemple) :

```
echo $ma_variable
```

renverra : unmot . Pour gérer les espaces et autres caractères spéciaux du shell, on utilisera les guillemets ou bien une notation avec des apostrophes :

```
echo $ma_variable

echo "$ma_variable"

echo ${ma_variable}

echo "${ma_variable} ${ma_variable2}"
```

enverront toutes la même réponse : unmot . Et avec des chemins de répertoires :

```
chemin_de_base="/home/username/un repertoire avec espaces"
chemin_complet="$chemin_de_base/repertoire"
```

Texte avec du style

In order to apply a style on your string, you can use a command like:

```
echo -e '\033[1mYOUR_STRING\033[0m'
```

Explanation:

- **echo -e** - The `-e` option means that escaped (backslashed) strings will be interpreted
- **\033** - escaped sequence represents beginning/ending of the style
- **lowercase m** - indicates the end of the sequence
- **1** - Bold attribute (see below for more)
- **[0m** - resets all attributes, colors, formatting, etc.

The possible integers are:

- **0** - Normal Style
- **1** - Bold
- **2** - Dim
- **3** - Italic
- **4** - Underlined
- **5** - Blinking
- **7** - Reverse
- **8** - Invisible

Les variables

Comme on le voit ci-dessus si on met une chaîne de caractères avec des espaces entre guillemets, la variable la prend bien mais attention à l'utiliser aussi avec des guillemets…

```
rsync -av "$chemin_complet" …
```

Sinon les espaces reprennent leurs rôles de séparateur!

Des variables système permettent d'accélérer la saisie et la compréhension.

Pour voir les variables d'environnement de votre système tapez simplement :

```
env
```

Quelques variables d'environnement à connaître : **HOME**, **USER**, **PATH**, **IFS**,… Pour appeler ou voir une variable, par exemple **HOME**, il suffit de mettre un **$** devant, par exemple :

```
echo $HOME
```

Ce petit code va afficher la variable **HOME** à l'écran.

Les variables

Il existe des variables un peu spéciales :

Nom	fonction
$*	contient tous les arguments passés à la fonction
$#	contient le nombre d'arguments
$?	contient le code de retour de la dernière opération
$0	contient le nom du script
$n	contient l'argument n, n étant un nombre
$!	contient le PID de la dernière commande lancée

Les variables

Exemple : créer le fichier arg.sh avec le contenu qui suit :

```
#!/bin/bash
echo "Nombre d'arguments ... :  "$#
echo "Les arguments sont ... :  "$*
echo "Le second argument est :  "$2

echo "Et le code de retour du dernier echo est :  "$?
```

Lancez ce script avec un ou plusieurs arguments et vous aurez :

```
./arg.sh 1 2 3
Nombre d'arguments ... :  3
Les arguments sont ... :  1 2 3
Le second argument est :  2
Et le code de retour du dernier echo est :  0
```

Exemple: un sleep interactif pour illustrer $!

Pour déclarer un tableau, plusieurs méthodes : première méthode (compatible bash, zsh, et ksh93 mais pas ksh88, ni avec dash, qui est lancé par "sh") :

```
tab=("John Smith" "Jane Doe")
```

ou bien :

```
tab[0]='John Smith'
tab[1]='Jane Doe'
```

Les variables

Pour compter le nombre d'éléments du tableau :

```
len=${#tab[*]} ou echo ${#tab[@]}
```

Pour afficher un élément :

```
echo ${tab[1]}
```

Pour afficher tous les éléments :

```
echo ${tab[@]}
```

Ou bien (en bash ou en ksh93 mais pas en ksh88) :

```
for i in ${!tab[@]}; do echo ${tab[i]}; done
```

Ou encore (C style) :

```
for (( i=0; i < ${#tab[@]}; i++ )); do echo ${tab[i]}; done
```

NB : toutes les variables sont des tableaux. Par défaut, c'est le premier élément qui est appelé :

```
echo ${tab[0]}
```

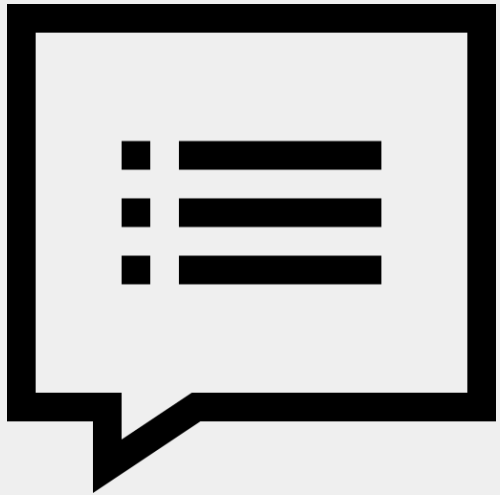
Et :

```
echo ${tab}
```

Renverront la même réponse. NB2 : les tableaux sont séparés par un séparateur défini : l'IFS. Par défaut l'IFS est composé des trois caractères : $' \t\n' soit espace, tabulation, saut de ligne. Il peut être forcé sur un autre caractère.

```
IFS=$SEPARATEUR
```

Les **arguments** en ligne de commande



Les arguments en ligne de commande

Pour passer des arguments en ligne de commande c'est encore une fois très simple.
Chaque argument est numéroté et ensuite on l'appelle par son numéro :

```
./test.sh powa noplay
```

Voici notre test.sh

```
#!/bin/sh
echo $3
echo $2
```

Notez que **$0** est le nom du fichier.

shift est une commande très pratique lorsque vous traitez des arguments en ligne de commande. Elle permet de faire "défiler" les arguments ($0, $1, $2, ...). C'est à dire que le contenu de **$1** passe dans **$0**, celui de **$2** dans **$1** et ainsi de suite. Il est tout à fait possible de traiter les arguments avec **for i in $*; do** mais lorsque vous aurez des options du style **–title "mon_titre"** il sera très laborieux de récupérer la valeur **"mon_titre"**.

Les arguments en ligne de commande

Voici un exemple de script où vous devez vous souvenir de ce que vous avez écrit (un petit jeu de mémoire) :

```
#!/bin/sh
clear # Un peu facile si la commande reste au dessus :-)
until [ $# = 0 ]
do
  echo -n "Taper l'option suivante : "
  read Reslt
  if [ "$Reslt" = "$1" ]; then
    echo "Bien joué !"
  else
    echo "Non mais quand même !!! C'ÉTAIT $1 ET NON PAS $Reslt PETIT FRIPON !!!"
    sleep 3 # Juste pour le fun du script qui rage ;-p
    echo "Donc je te bannis ! Et toc !! Tu ne peux rien contre moi !!!"
    exit 1
  fi
  shift # On défile
done
echo "Vous avez réussi !"
```

Variables prépositionnées

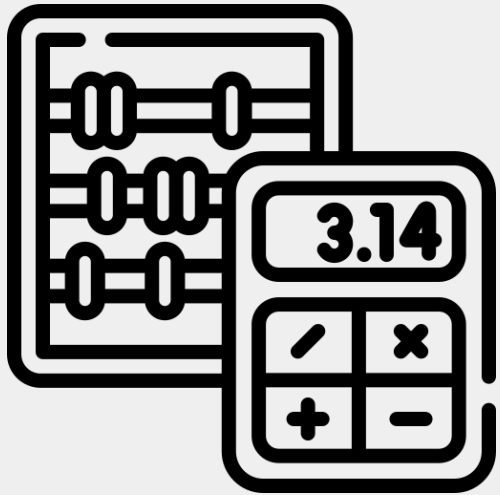
Certaines variables ont une signification spéciale réservée. Ces variables sont très utilisées lors la création de scripts :

- pour récupérer les paramètres transmis sur la ligne de commande,
- pour savoir si une commande a échoué ou réussi,
- pour automatiser le traitement de tous paramètres.

Liste de variables prépositionnées

- **$0** : nom du script. Plus précisément, il s'agit du paramètre 0 de la ligne de commande, équivalent de argv[0]
- **$1, $2, ..., $9** : respectivement premier, deuxième, ..., neuvième paramètre de la ligne de commande
- **$*** : tous les paramètres vus comme un seul mot
- **$@** : tous les paramètres vus comme des mots séparés : "$@" équivaut à "$1" "$2" ...
- **$#** : nombre de paramètres sur la ligne de commande
- **$-** : options du shell
- **$?** : code de retour de la dernière commande
    - **$$** : PID du shell
- **$!** : PID du dernier processus lancé en arrière-plan
- **$_** : dernier argument de la commande précédente

L'arithmétique



L'arithmétique

```
(( variable = 2 + $autre_var * 5 ))
```

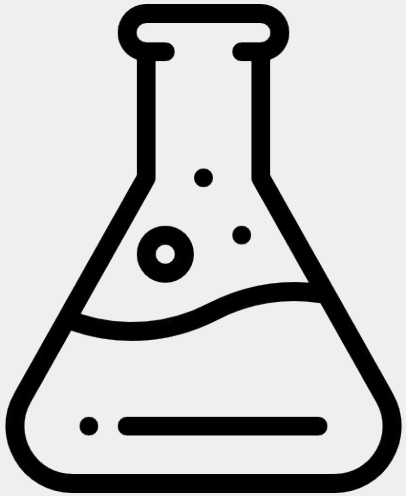
Exemple : besoin de définir des plages de valeurs (1 à 500 puis 501 à 1000 puis 1001 à 1500...)

```
id_per_step = 500
for (( i=0; i<8; i++ )); do
    (( min_step_id = 1 + $i * $id_per_step ))
    (( max_step_id = (( $i + 1 )) * $id_per_step ))
    echo "$min_step_id to $max_step_id "
done
```

Exercices : Shell Allergies

Cliquez ici pour réaliser l’exercice

La commande test



Opérateurs de test sur fichiers

La commande test existe sous tous les Unix, elle permet de faire un test et de renvoyer 0 si tout s'est bien passé ou 1 en cas d'erreur.

En mode console, faites **man test** pour connaître tous les opérateurs, en voici quelques-uns.

Syntaxe	Fonction réalisée
`-e fichier`	renvoie 0 si fichier existe.
`-d fichier`	renvoie 0 si fichier existe et est un répertoire.
`-f fichier`	renvoie 0 si fichier existe et est un fichier 'normal'.
`-w fichier`	renvoie 0 si fichier existe et est en écriture.
`-x fichier`	renvoie 0 si fichier existe et est exécutable.
`f1 -nt f2`	renvoie 0 si f1 est plus récent que f2.
`f1 -ot f2`	renvoie 0 si f1 est plus vieux que f2.

Opérateurs de comparaison numérique

Syntaxe	Fonction réalisée
$A -lt 5	renvoie 0 si $A est strictement inférieur à 5
$A -le 5	renvoie 0 si $A est inférieur ou égal à 5
$A -gt 5	renvoie 0 si $A est strictement supérieur à 5
$A -ge 5	renvoie 0 si $A est supérieur ou égal à 5
$A -eq 5	renvoie 0 si $A est égal à 5
$A -ne 5	renvoie 0 si $A est différent de 5

Les **crochets**

Les arguments en ligne de commande

On peut raccourcir la commande test par des crochets. Exemple :

```
test -f /etc/passwd
echo $?
0
[ -f /etc/passwd ]
echo $?
0
```

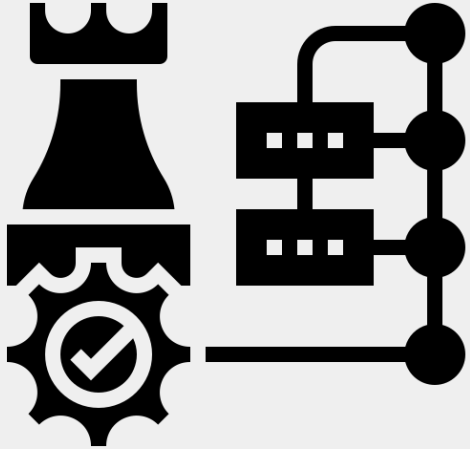
Affichera la valeur 0 : ce fichier existe, 1 dans le cas où le fichier **/etc/passwd** n'existe pas. Sous Unix, le code de retour est par convention et en général **0** s'il n'y a aucune erreur et différent de 0 dans les autres cas. La syntaxe la plus appropriée dans de la programmation shell moderne est le double crochet :

```
[[ -f /etc/passwd ]]
```

Cela gère bien mieux les problèmes d'espaces dans les noms de fichiers, les erreurs etc...

C'est une structure propre à bash (ksh, ?) qui est le shell par défaut dans la plupart des distributions Linux, et de Ubuntu en particulier. On garde en général des simples crochets pour les scripts shell qui doivent être à tout prix **POSIX** (utilisation sur des Unix sans installation préalable de bash, comme BSD, Solaris...) .

Les **opérateurs logiques**



Les arguments en ligne de commande

Il y a en 3 :

- le **et** logique : -a
- le **ou** logique : -o
- le **non** logique : !

Exemple :

```
echo "renverra 0 si les deux expressions sont vraies"
test expr1 -a expr2
[ expr1 -a expr2 ]
```

Table de vérité de « **-a** »

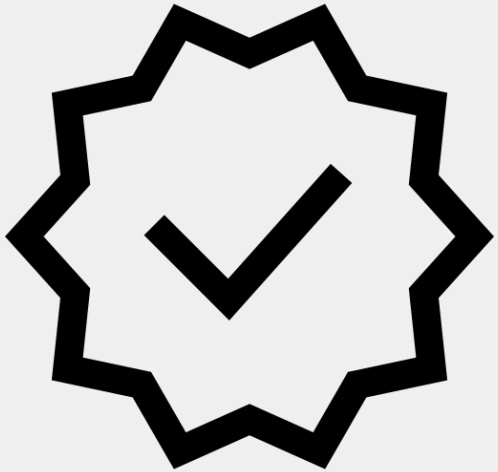


Table de vérité de « -a » :

Comparaison	Résultat	Calcul
0 et 0	0	0 × 0 = 0
0 et 1	0	0 × 1 = 0
1 et 0	0	1 × 0 = 0
1 et 1	1	1 × 1 = 1

Les deux assertions doivent être vérifiées pour que la condition le soit aussi.

Table de vérité de « **-o** »

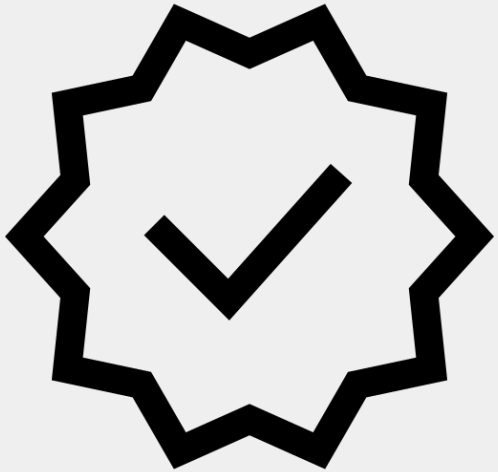


Table de vérité de « -o »

Comparaison	Résultat	Calcul
0 ou 0	0	0 + 0 = 0
0 ou 1	1	0 + 1 = 1
1 ou 0	1	1 + 0 = 1
1 ou 1	1	1 + 1 = 1

Dès que l'une des deux assertions est vérifiée, la condition globale l'est aussi.

Exemple plus complet :

```
#!/bin/sh

echo -n "Entrez un nom de fichier: "
read file
if [ -e "$file" ]; then
       echo "Le fichier existe!"
else
       echo "Le fichier n'existe pas, du moins n'est pas dans le répertoire

 d'exécution du script"
```

*La seule chose qui prête à confusion est que l'on vérifie seulement si le fichier **« file »** est dans le répertoire où le script a été exécuté.*

La structure : `if`

La structure : `if`

Avant de commencer à faire des scripts de 1000 lignes, il serait intéressant de voir comment se servir des variables, et des instructions **if, then, elif, else, fi**.

Cela permet par exemple de faire réagir le script de manière différente, selon la réponse de l'utilisateur à une question.

En bash, les variables ne se déclarent généralement pas avant leur utilisation, on les utilise directement et elles sont créées lors de sa première mise en œuvre. Pour pouvoir voir la valeur d'une variable il faut faire précéder son nom du caractère **« $ »**.

```
#!/bin/sh
echo -n "Voulez-vous voir la liste des fichiers Y/N : "
read ouinon
if [ "$ouinon" = "y" ] || [ "$ouinon" = "Y" ]; then
    echo "Liste des fichiers :"
    ls -la
elif [ "$ouinon" = "n" ] || [ "$ouinon" = "N" ]; then
    echo "Ok, bye! "
else
    echo "Il faut taper Y ou N!! Pas $ouinon"
fi
```

Explication

Ce script peut paraître simple à première vue mais certaines choses prêtent à confusion et ont besoin d'être expliquées en détail.

Tout abord, le **`echo -n`** permet de laisser le curseur sur la même ligne, ce qui permet à l'utilisateur de taper la réponse après la question (question d'esthétique).

L'instruction **`read`** permet d'affecter une valeur ou un caractère à une variable quelconque, en la demandant à l'utilisateur.

En bash, la variable est considérée comme une chaîne même si celle-ci contient une valeur numérique, et les majuscules sont considérées différentes des minuscules, $M ≠ $m.

Ensuite vient l'instruction conditionnelle **`if`**. Elle est suivie d'un **« [»** pour délimiter la condition. La condition doit bien être séparée des crochets par un espace ! Attention, la variable est mise entre guillemets car dans le cas où la variable est vide, le shell ne retourne pas d'erreur, mais en cas contraire, l'erreur produite ressemble à :

```
[: =: unaryoperator expected
```

Explication

L'opérateur `**||**` signifie exécuter la commande suivante si la commande précédente n'a pas **renvoyé 0**. Il existe aussi l'opérateur **&&** qui exécute la commande suivante si la commande précédente a **renvoyé 0**, et enfin **;** qui exécute l'opération suivante dans tous les cas.

Exemple, créer le répertoire toto s'il n' existe pas :

```
[ ! -d /tmp/toto ] && mkdir /tmp/toto
[ -d /tmp/toto ] || mkdir /tmp/toto
test ! -d /tmp/toto && mkdir  /tmp/toto
rm -rf /tmp/toto;mkdir /tmp/toto
```

Les **« { »** servent à bien délimiter le bloc d'instructions suivant le `**then**`, est une commande et donc si elle est sur la même ligne que le `**if**` les deux commandes doivent être séparées par un `**;**`

Ensuite, `**elif**` sert à exécuter une autre série d'instructions, si la condition décrite par `**if**` n'est pas respectée, et si celle fournie après ce `**elif**` l'est.

Enfin, `**else**` sert à exécuter un bloc si les conditions précédentes ne sont pas respectées.

`**fi**` indique la fin de notre bloc d'instructions `**if**`. Cela permet de voir où se termine la portion de code soumise à une condition.

Exemples de quelques commandes pratiques

```
sh -n nom_du_fichier
```

ou

```
bash -x chemin_du_fichier
```

Cette commande vérifie la syntaxe de toutes les commandes du script, pratique quand on débute et pour les codes volumineux.

```
sh -u nom_du_fichier
```

Celle-ci sert à montrer les variables qui n'ont pas été utilisées pendant l'exécution du programme. Voici le tableau des opérateurs de comparaison, ceux-ci peuvent s'avérer utiles pour diverses raisons, nous verrons un peu plus loin un exemple.

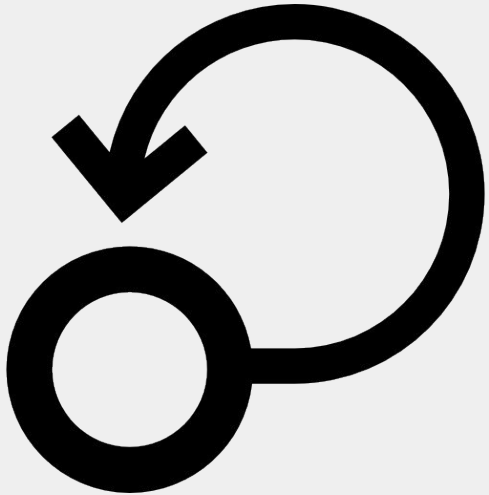
```
$A = $B # Vérifie si les deux chaînes sont égales.

$A != $B # Vérifie si les deux chaînes sont différentes.

-z $A # Vérifie si A n'existe pas (ne contient pas de chaîne).

-n $A # Vérifie si A existe (contient une chaîne).
```

Les structures **while & until**



Les structures while et until

La commande while exécute ce qu'il y a dans son bloc tant que la condition est respectée :

```
#!/bin/sh

cmpt=1
cm=3
echo -n "Mot de passe : "
read mdp

while [ "$mdp" != "ubuntu" ] && [ "$cmpt" != 4 ]
do
     echo -n "Mauvais mot de passe, plus que "$cm" chance(s): "
     read mdp
     cmpt=$(($cmpt+1))
     cm=$(($cm-1))
done
echo "Non mais, le brute-force est interdit en France !!"
```

On retrouve des choses déjà abordées avec `**if**`. Le `**&&**` sert à symboliser un **"et"**, cela implique que deux conditions sont à respecter. Le `**do**` sert à exécuter ce qui suit si la condition est respectée. Si elle ne l'est pas, cela saute tout le bloc (jusqu'à `**done**`). Vous allez dire :

Mais qu'est-ce que c'est ce truc avec cette syntaxe bizarre au milieu ?

Les structures while et until

Cette partie du code sert tout simplement à réaliser une opération arithmétique.
A chaque passage, **'cmpt = cmpt+1' et 'cm = cm-1'**.

`while` permet de faire exécuter la portion de code un nombre indéterminé de fois. La commande **`until`** fait la même chose que la commande **`while`** mais en inversant. C'est-à-dire qu'elle exécute le bloc jusqu'à ce que la condition soit vraie, donc elle s'emploie exactement comme la commande **`while`**.

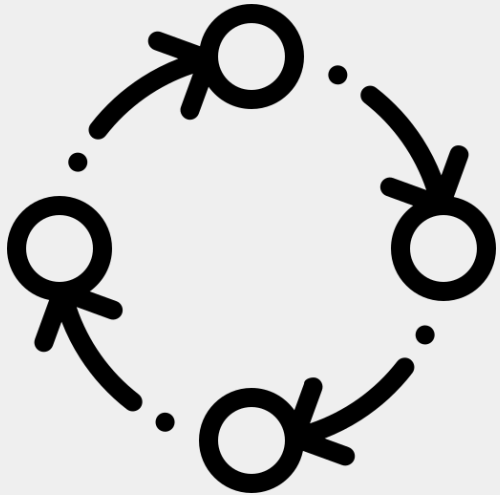
Par exemple, si on a besoin d'attendre le démarrage de notre window manager pour exécuter des commandes dans notre Xsession il sera plus intéressant d'utiliser le **`until`**:

```
#!/bin/sh
until pidof wmaker
do
  sleep 1
done
xmessage "Session loaded" -buttons "Continue":0,"That all":1;
[ $? -eq 0 ] && xmessage "Load more..."
```

Mais on aurait pu aussi faire:

```
#!/bin/sh
while [ -z $(pidof wmaker) ]
do
 sleep 1
done
#(...)
```

La structure case



La structure case

Regardons la syntaxe de cette commande, qui n'est pas une des plus simples :

```
case variable in
    modèle [ | modèle] ...) instructions;;
    modèle [ | modèle] ...) instructions;;
    ...
esac
```

Cela peut paraître complexe mais on s'y habitue quand on l'utilise. Mais à quoi sert cette commande ?

Elle sert à **comparer le contenu d'une variable à des modèles différents**.
Les **;;** sont indispensables car il est possible de placer plusieurs instructions entre un modèle et le suivant.
Les **;;** servent donc à identifier clairement la fin d'une instruction et le début du modèle suivant.

Exemple :

```
#!/bin/sh
echo -n "Êtes-vous fatigué ? "
read on
case "$on" in
    oui | o | O | Oui | OUI ) echo "Allez faire du café !";;
    non | n | N | Non | NON ) echo "Programmez !";;
    * ) echo "Ah bon ?";;
esac
```

La seule chose qui mérite vraiment d'être expliquée est sans doute **`*)`**. Cela indique tout simplement l'action à exécuter si la réponse donnée n'est aucune de celles données précédemment.

Les **fonctions**

Les fonctions

Les fonctions sont indispensables pour bien structurer un programme mais aussi pouvoir le simplifier, créer une tâche, la rappeler... Voici la syntaxe générale de 'déclaration' d'une fonction :

```
nom_fonction(){
  instructions
}
```

Cette partie ne fait rien en elle même, elle dit juste que quand on appellera nom_fonction, elle fera instruction. Pour appeler une fonction (qui ne possède pas d'argument, voir plus loin) rien de plus simple :

```
nom_fonction
```

Rien ne vaut un petit exemple :

```
#!/bin/sh

#Definition de ma fonction
mafonction(){
  echo 'La liste des fichiers de ce répertoire'
  ls -l
}
#fin de la définition de ma fonction

echo 'Vous allez voir la liste des fichiers de ce répertoire:'
mafonction       #appel de ma fonction
```

Les fonctions

Comme vous l'avez sans doute remarqué, quand on appelle la fonction, on exécute simplement ce qu'on lui a défini au début, dans notre exemple, echo... et **ls -l**, on peut donc faire exécuter n'importe quoi à une fonction. Les fonctions peuvent être définies n'importe où dans le code du moment qu'elles sont définies avant d'être utilisées. Même si en bash les variables sont globales, il est possible de les déclarer comme locales au sein d'une fonction en la précédant du mot clé local: local **ma_fonction**.

Exemple: un sleep interactif :

```
#!/bin/bash
info(){
        echo -e "$1\nBye"
        exit
}
test -z "$1" && info "requiert 1 argument pour le temps d'attente..." || PRINT=$(($1*500))
test -z $(echo "$1" | grep -e "^[0-9]*$") && info "'$1' est un mauvais argument"
test $1 -gt 0 || info "Je ne prends que les entiers > 0"
print_until_sleep(){
         local COUNT=0
         while [ -d /proc/$1 ]; do

                 test $(($COUNT%$2)) -eq 0 && echo -n "*"
                 COUNT=$(($COUNT+1))
         done
}
sleep $1 & print_until_sleep $! $PRINT
echo -e "\nBye"
```

Les fonctions

Vous l'aurez compris en regardant l'exemple avec la fonction info vu plus haut avec la magie de la fonction c'est que l'on peut lui passer des paramètres comme les paramètres que vous passez à votre script.

Souvenez-vous **$1 $2 $3 $n** (**$0** pour le nom du script) et (**$#** le nombre de paramètre passé au script) ...

Une fonction comprend plusieurs instructions. Vous pouvez l'appeler autant de fois que vous voulez dans votre code : laissez donc votre fonction assez générique et faites-la varier grâce aux paramètres que vous pouvez préciser en l'appelant.

Vous pouvez déclarer une fonction Bash avec **maFonction()** ou function **maFonction**. Vous faites ensuite appel à votre fonction **maFonction()** en tapant simplement dans votre code **maFonction.**

Structure de base d’un script bash

Quel serait la structure de base d’un script Bash ?

- 1 / **Shebang**
- 2 / **Commentaires**
- 3 / **Fonction gestion de la syntaxe**
- 4 / **Fonction(s) utile(s)**
- 5 / **Corps principal**
- 6 / **Fin**

```
#!/bin/bash
# main.sh structure de base d’un script

target=$1
usage() {
    echo "Usage: $0 <fichier>"
    echo "Compte les lignes d'un fichier"
    exit
}

main() {
    ls -l $target
    echo "nombre de lignes : $(wc -l $target)"
    stat $target
}

if [ $# -lt 1 ]; then
    usage
elif [ $# -eq 1 ]; then
    main
else
    usage
fi
 exit
```

Evaluation : My Magic Prompt

Réaliser un prompt en bash script

Evaluation : My Magic Prompt

Pour accéder au prompt vous devrez faire en sorte d'entrer un **mot de passe** et un **login** spécifique. L'intégralité du projet devra être réaliser dans un dossier **`~/my-magic-prompt/main.sh`**

Ce **prompt** devra comporter les commandes de base suivantes :

- **`help`** : qui indiquera les commandes que vous pouvez utiliser
- **`ls`** : lister des fichiers et les dossiers visible comme caché
- **`rm`** : supprimer un fichier
- **`rmd`** ou **rmdir** : supprimer un dossier
- **`about`** : une description de votre programme
- **`version`** ou **`--v`** ou **`vers`** : affiche la version de votre prompt
- **`age`** : vous demande votre âge et vous dit si vous êtes majeur ou mineur
- **`quit`** : permet de sortir du prompt
- **`profil`** : permet d'afficher toutes les informations sur vous même.
  - *First Name, Last name, age, email*
- **passw** : permet de changer le password avec une demande de confirmation
- **`cd`** : aller dans un dossier que vous venez de créer ou de revenir à un dossier précédent
- **`pwd`** : indique le répertoire actuelle courant
- **`hour`** : permet de donner l'heure actuelle
- **`*`** : indiquer une commande inconnu
- **`httpget` :** permet de télécharger le code source html d'une page web et de l'enregistrer dans un fichier spécifique. Votre prompt doit vous demander quel sera le nom du fichier.
- **smtp** : vous permet d'envoyer un mail avec une adresse un sujet et le corp du mail
- **open :** ouvrir un fichier directement dans l'éditeur VIM même si le fichier n'existe pas

*Pour faire valider votre exercice, **il devra être sur gitlabs** …*

Vous devez réaliser la documentation du prompt dans un README.md avec markdown

*Vous devrez faire en sorte que **le prof valide votre travail**. Vous trouverez le début du tp **ici***

*Si vous aidez votre prochain vous pouvez gagner **1 point bonus**.*

*Vous devez respecter **"la structure de base d'un script Bash"** vu dans ce cours.*

Evaluation BONUS : My Magic Prompt

Ajouter la commande :

- **rps** : permet de jouer à 2 joueurs à **Rock Paper Scissors**
  - Faite en sorte que deux joueurs puisse jouer
    - Demander le nom du P1 et le nom du P2
  - Faite en sorte que le prompt affiche le nom du joueur qui doit jouer au tour par tour
  - Faite en sorte de déterminer un vainqueur en 3 manches
  - Faite en sorte de compter les points
  - Faite en sorte de créer un pouvoir ultime qui permet de gagner à tous les coups (SuperKitty)

- **rmdirwtf :** permet de supprimer un ou plusieurs dossier
  - Cette commande doit être protéger par un mot de passe et prendre en compte le changement du mot de passe