

# Scripts avec bash

---

Guillaume Chanel - [guillaume.chanel@hesge.ch](mailto:guillaume.chanel@hesge.ch)

Florent Glück - [florent.gluck@hesge.ch](mailto:florent.gluck@hesge.ch)

April 02, 2025

ISC - HEPIA

# Remerciements

Remerciements à Jean-Luc Falcone

# Scripts

---

# Problème réel

- Aggréger des listes d'emails
- Supprimer les doublons
- Les adresses peuvent être écrites avec des casses différentes

# Solution possible

À l'aide des commandes shell suivantes :

<b>cat</b>	Concatène plusieurs fichiers
<b>tr</b>	Transforme les caractères
<b>sort</b>	Trie les lignes
<b>uniq</b>	Rend unique les lignes répétées

Exemple :

```
$ cat mail1.txt mail2.txt | tr '[A-Z]' '[a-z]' | sort | uniq > result.txt
```

# Qu'est ce qu'un script ?

- Un script est un fichier texte contenant une séquence de commandes shell
- Un script peut être exécuté comme une commande ou un programme

# Exécuter un script

Deux manières d'exécuter un script :

- Dans le shell **actuel**, avec la commande **source** :

```
$ source backup.sh
```

- Dans un **nouveau** shell (processus)

- passé en argument à l'exécution d'un shell :

```
$ bash backup.sh
```

- en lui ajoutant le bit d'exécution, puis en l'exécutant :

```
$ chmod u+x backup.sh      # à ne faire qu'une seule fois  
$ ./backup.sh
```

# `#!` Hash bang - spécifier l'interpréteur

- On peut préciser l'interpréteur du script à exécuter avec les caractères `#!` au début du script (prononcer *hash-bang*, *she-bang* ou *sha-bang*)
- Ceci s'applique :
  - à **tout** interpréteur, e.g. bash, zsh, python, perl, lua, ruby, etc.
  - **uniquement** lorsque le script est exécuté directement, e.g. :

```
$ ./backup.sh
```



# #! Hash bang : exemples

Script bash, `uid.sh` :

```
#!/bin/bash  
echo $UID
```

Script python, `uid.py` :

```
#!/usr/bin/python3  
import os  
print(os.getuid())
```

Script ruby, `uid.rb` :

```
#!/usr/bin/ruby  
puts Process.uid
```

# Exercice : premier script

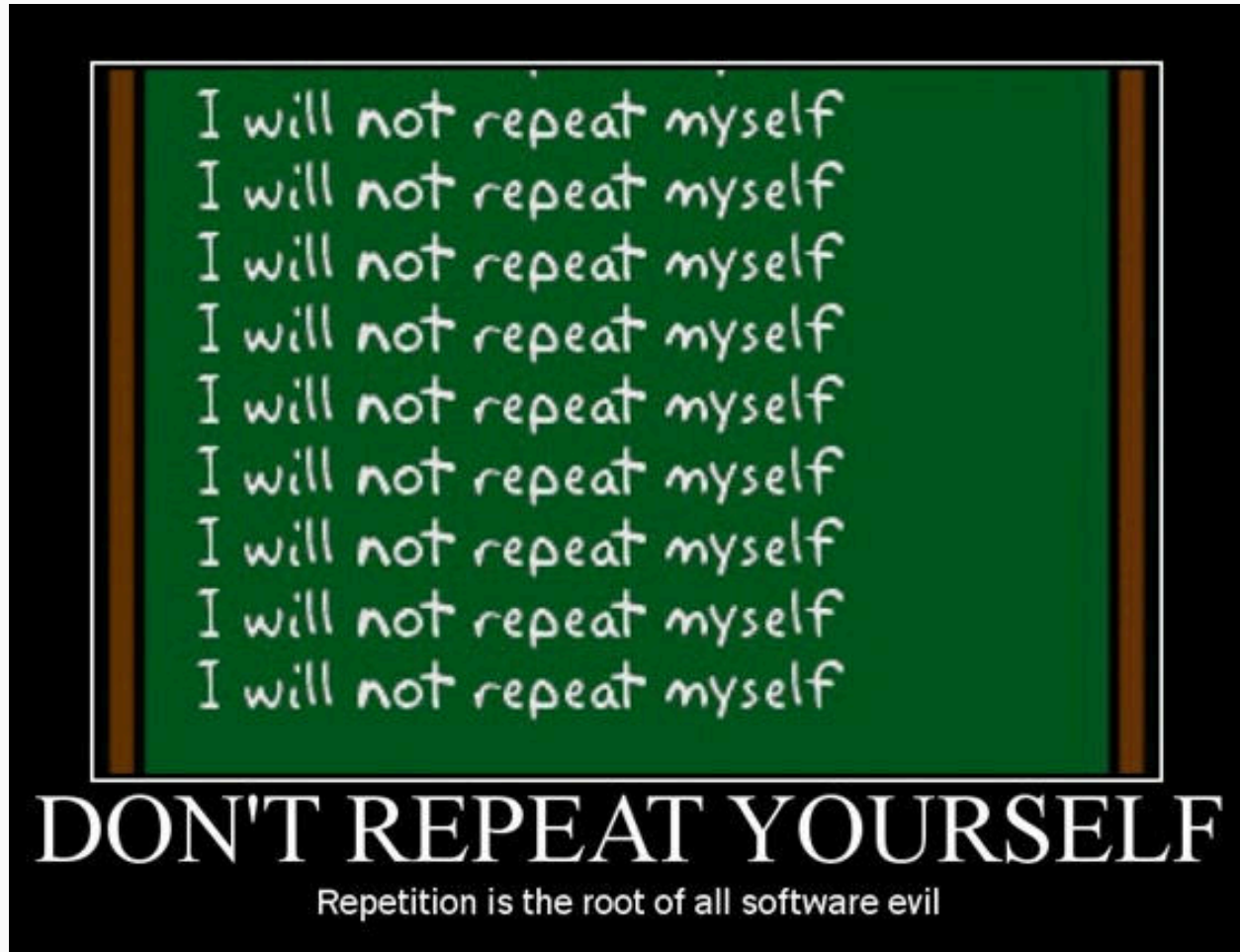
Écrire un script qui :

- Supprime le contenu du dossier `data/backup`
- Copie tous les fichiers `*.txt` et `*.dat` du dossier `data` vers le dossier de backup (voir architecture `git`)
- Change la permission des fichiers copiés pour ne donner QUE les droits de lecture à l'utilisateur et au groupe

# Solution

backup1.sh :

```
#!/bin/bash
# Efface les anciens backups
rm -f ~/data/backup/*
# Backup les fichiers .txt et .dat
cp ~/data/*.txt ~/data/*.dat ~/data/backup/
chmod 440 ~/data/backup/*
```



# Variables

- On assigne des variables avec le symbole =
- On accède à la valeur de la variable avec le symbole \$
- Exemple :

```
$ message="Je suis ... "  
$ echo message  
$ echo $message  
$ echo ${message}  
$ echo "${message}Luke"
```

**Attention** : les variables déclarées ainsi sont **globales**, même si elles sont déclarées dans le corps d'une fonction ou d'une boucle !

## Exercice : variables

Modifiez le script précédent afin d'éviter les répétitions

# Exercice : variables

Modifiez le script précédent afin d'éviter les répétitions

Solution, `backup2.sh` :

```
#!/bin/bash
DATA=~/.data
BACKUP=$DATA/backup
#Efface les anciens backups
rm -f $BACKUP/*
#Backup les fichiers .txt et .dat
cp $DATA/*.txt $DATA/*.dat $BACKUP
chmod 440 $BACKUP/*
```

# Code de retour

Toute commande retourne un code entier positif avec pour valeur :

- **0** : si la commande c'est terminée avec succès (équivalent à VRAI)
- **code d'erreur**  $> 0$  : si la commande a rencontré une erreur (équivalent à FAUX)

La variable `$?` contient le code de retour de la dernière commande :

```
$ ls *.sh  
$ RESULT=$?
```



Ne pas **confondre** le **code de retour** d'une commande avec sa **sortie standard**, par exemple dans les substitutions !



# Code de retour et enchaînement de commandes

Les listes ET/OU permettent d'enchaîner les commandes :

- Ici, `process` est exécuté, uniquement si `ls` retourne un code de 0 (vrai) :

```
$ ls fichierExistant && process fichierExistant
```

- Ici, `echo` est exécuté, uniquement si `ls` retourne un code différent de 0 (donc une erreur = faux) :

```
$ ls fichierNonExistant || echo "Erreur le fichier n'existe pas" 1>&2
```



La valeur du code de retour est interprétée à l'inverse du langage C (0 pour vrai et  $> 0$  pour faux)!

# Conditions (if then)

La structure de contrôle `if` existe comme dans la plupart des langages :

```
if command; then
    instruction1
    instruction2
    ...
fi
```

```
if command; then instruction1 ; instruction2 ; ... ; fi
```

# Conditions (if then else)

```
if command; then
  instruction1
  instruction2
  ...
else
  instruction5
  instruction6
  ...
fi
```

# Conditions (if then elif else)

```
if command; then
    instruction1
    instruction2
    ...
elif command; then
    instruction5
    instruction6
    ...
else
    instruction10
    instruction11
    ...
fi
```

# Les tests

- On peut tester une expression avec la commande `test` ou avec des crochets
- Les deux expressions ci-dessous sont équivalentes<sup>1</sup> :

```
test EXPRESSION
```

```
[ EXPRESSION ]
```

- `test` et `[` retournent le résultat du test sous forme de code de retour

---

<sup>1</sup>Les programmes `test` et `[` se comportent de manière presque identique

# Tests sur les fichiers

<b>-e FILE</b>	Le fichier FILE existe
<b>-f FILE</b>	Le fichier FILE existe et est un fichier régulier
<b>-d FILE</b>	Le fichier FILE existe et est un répertoire
<b>-x FILE</b>	Le fichier FILE existe et est exécutable
<b>-w FILE</b>	Le fichier FILE existe et on peut y écrire
<b>FILE1 -nt FILE2</b>	Le fichier FILE1 est plus récent que le fichier FILE2
<b>FILE1 -ot FILE2</b>	Le fichier FILE1 est plus ancien que le fichier FILE2

# if : exemples

Est-ce que le fichier `/dev/urandom` existe et est un fichier régulier ?

```
if test -f /dev/urandom; then
    echo YES
else
    echo NO
fi
```

```
if [ -f /dev/urandom ]; then
    echo YES
else
    echo NO
fi
```

## Exercice : if

Modifiez le script précédent pour que :

- Si le répertoire `backup` existe, son contenu soit effacé (comme avant)
- Si le répertoire `backup` n'existe pas, il soit créé

Assurez-vous que votre script fonctionne correctement dans le cas où le nom de répertoire est le même que celui d'un fichier



# Solution

backup3.sh :

```
...  
DATA=~ /data  
BACKUP=$DATA/backup  
# Si repertoire existe  
if [ -d $BACKUP ]; then  
    # Efface les anciens backups  
    rm -f $BACKUP/*  
else  
    # Cree repertoire  
    mkdir -p $BACKUP  
fi  
...
```

# Autres expressions de test

<b>! EXPRESSION</b>	Retourne vrai si EXPRESSION est fausse
<b>EXPR1 -a EXPR2</b>	Retourne vrai si EXPR1 <b>et</b> EXPR2 sont vraies
<b>EXPR1 -o EXPR2</b>	Retourne vrai si EXPR1 <b>ou</b> EXPR2 sont vraies
<b>STRING1 = STRING2</b>	Retourne vrai si STRING1 est égal à STRING2
<b>STRING1 != STRING2</b>	Retourne vrai si STRING1 n'est pas égal à STRING2

# Boucles (for) (1/2)

- Itère sur chaque éléments d'une liste
- Les éléments sont séparés par des espaces ou des retours à la ligne
- Assigne une variable utilisable à l'intérieur de la boucle

```
for VAR in LIST; do  
    CMD1 $VAR  
    CMD2 $VAR  
    ...  
done
```

# Boucles (for) (2/2)

Une liste peut être :

- Des éléments séparés par des espaces :

```
for ANIMAL in chat chien oiseau
```

- Un intervalle :

```
for I in {5..42}
```

- Une expression représentant un ensemble de fichiers :

```
for FILE in *.txt
```

## Exercice : for

Modifiez le script précédent avec une boucle `for` qui pour chaque fichier :

- Copie le fichier
- Affiche un message d'information de copie

Le mode des fichiers doit ensuite être changé pour tous les fichiers en une fois

# Solution

backup4.sh :

```
...  
DATA=~ /data  
BACKUP=$DATA/backup  
# Backup les fichiers .txt et .dat  
for f in $DATA/*.txt $DATA/*.dat; do  
    echo "Copie de $f"  
    cp $f $BACKUP  
done  
chmod 440 $BACKUP/*  
...
```

# Substitution de commande

- Il est souvent intéressant de récupérer le résultat d'une commande
- Deux formes de syntaxes existent pour cela :
  - `$(CMD OPTIONS ARGUMENTS)`
  - ``CMD OPTIONS ARGUMENTS``
- Exemple :

```
$ echo "Mon nom d'utilisateur est: $(whoami)"  
$ echo "Mon nom d'utilisateur est: `whoami`"  
$ a=$(ls *.sh)  
$ a=`ls *.sh`
```

## Exercice : substitution de commande

Comment changer le code précédent pour effectuer le changement de mode dans la boucle (indice : commande `basename`) ?



# Exercice : substitution de commande

Comment changer le code précédent pour effectuer le changement de mode dans la boucle (indice : commande `basename`) ?

Solution, `backup5.sh` :

```
...  
DATA=~ /data  
BACKUP=$DATA/backup  
# Backup les fichiers .txt et .dat  
for f in $DATA/*.txt $DATA/*.dat; do  
    echo "Copie de $f"  
    cp $f $BACKUP  
    name=$(basename $f)  
    chmod 440 $BACKUP/$name  
done  
...
```

# Arguments

Il est possible de récupérer les arguments passés à un script en utilisant des **variables prédéfinies** :

<b>\$0</b>	Contient le nom du script tel qu'il a été appelé (e.g. /home/jane/script.sh ou ./script.sh)
<b>\$1, \$2, \$3, etc.</b>	\$1 indique le premier argument passé au script, \$2 le deuxième, etc.
<b>\$@</b>	La liste des arguments (sans le nom du script)
<b>\$#</b>	Le nombre d'arguments passés au script

## Exercice : arguments

Modifiez le script précédent pour que les répertoires source et destination puissent être passés en argument au script

## Exercice : arguments

Modifiez le script précédent pour que les répertoires source et destination puissent être passés en argument au script

Solution, `backup6.sh` :

```
#!/bin/bash  
DATA=$1  
BACKUP=$2  
...
```

Utilisation :

```
$ ./backup6.sh ~/data ~/data/backup
```

# Fonction (1/2)

Une fonction se définit de la manière suivante :

```
function show {  
    # $1 et $2 sont les 1er et 2eme arguments passés à la fonction  
    # $@ sont tous les arguments passés à la fonction  
    echo "$1 ----- $2"  
}  
show foo bar # appel à la fonction
```

- On peut accéder aux arguments d'une fonction avec `$1`, `$2`, `$@`, etc.
- Ceux-ci *override* les arguments du script

## Fonction (2/2)



Une fonction n'a **pas de valeur de retour** !

- Cependant, il est possible de retourner des valeurs en utilisant :
  - des variables globales
  - la sortie standard en combinaison avec la substitution de commande :

```
function show {  
    # $1 et $2 sont les 1er et 2eme arguments passés à la fonction  
    # $@ sont tous les arguments passés à la fonction  
    echo "$1 ----- $2"  
}  
  
RET=$(show foo bar) # appel à la fonction  
echo $RET
```

# Exercice : fonctions

- Dans le script précédent, remplacez les trois opérations du backup par un appel à une fonction avec arguments
- Incluez également le changement de mode dans la boucle

# Solution

backup7.sh :

```
function copy {  
    echo "Copie de $1"  
    cp $1 $2  
    name=$(basename $1)  
    chmod 440 $2/$name  
}  
...  
for f in $DATA/*.txt $DATA/*.dat; do  
    copy $f $BACKUP # appel à la fonction  
done  
...
```



# Exercice : script final

Modifiez le script précédent afin qu'il :

- Vérifie qu'il est appelé avec exactement deux arguments
  - sinon, affiche un message d'aide indiquant la syntaxe sur `stderr` et se termine (avec un code d'erreur)
- Vérifie la validité des arguments (répertoires existants)
  - sinon, affiche un message d'erreur sur `stderr` et se termine (avec un code d'erreur)
- Affiche un message d'erreur sur `stderr` pour toute erreur rencontrée (et ne réalise pas d'opération indésirable)
- Renvoie un code de retour de 0 si tout c'est bien passé

## Exercice : source

- Écrivez le script `var.sh` qui définit quelques variables
- Exécutez le script créé (soit avec `bash var.sh`, soit avec `./var.sh`)
- Affichez les valeurs des variables définies précédemment : est-ce qu'elles existent ?
- Plutôt que d'exécuter le script créé, exécutez `source var.sh`
- Affichez les valeurs des variables définies précédemment : est-ce qu'elles existent ?
- Qu'en concluez-vous ?

# Variables d'environnement

---

# Que sont les variables d'environnement ?

- Variables normales uniquement accessibles au processus courant
- Variables d'environnement accessibles à une hiérarchie de processus
- Les processus enfants héritent des variables d'environnement du parent
- On accède à leur valeur avec un `$` (comme une variable normale)
- On les définit avec le mot clé **export** :

```
$ export VAR1="hello world"  
$ echo $VAR
```

- Tous les langages de programmation sérieux permettent d'y accéder

# Exercice : variables d'environnement

1. Dans le shell courant, créez la variable normale `houba`
2. Exécutez un nouveau shell
3. Dans le nouveau shell, est-ce que la variable `houba` existe ?
4. Effectuez les trois opérations ci-dessus mais avec une variable d'environnement
5. Qu'en concluez-vous ?

# Variables d'environnement standard

<b>\$PATH</b>	Chemin où trouver les fichiers exécutables
<b>\$HOME</b>	Répertoire <i>home</i> de l'utilisateur
<b>\$LANG</b>	Langue par défaut
<b>\$DISPLAY</b>	Serveur graphique
<b>\$PS1</b>	<i>Prompt</i> (invite) du shell
<b>\$HOSTNAME</b>	Nom de la machine

# Chemin (PATH)

- La variable d'environnement **PATH** définit l'ensemble des répertoires où trouver les fichiers exécutables (programmes/scripts)
- Si l'exécutable ne se trouve pas dans un des répertoires du **PATH**, alors le chemin complet doit être précisé
- Exemple de **PATH** :

```
/home/jane/bin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

- Ligne typique dans la configuration de bash (fichier `~/.bashrc`) :

```
export PATH=~/.local/bin:$PATH
```

# Exercice PATH

1. Créez l'arborescence `.local/bin/` dans votre *home* et ajoutez-y un script exécutable
2. Tentez d'exécuter ce script sans indiquer son chemin
3. Ajoutez ce chemin à votre `PATH`
4. Vérifiez que cela fonctionne en exécutant le script, sans indiquer son chemin et depuis un autre répertoire



# Permissions par défaut

---

# umask

- Il est possible de spécifier un masque de permissions par défaut pour la création de fichiers et répertoires grâce à la commande `umask` :

```
umask [permissions]
```

- Ci-dessus, `permissions`, représente les permissions **à retirer** en octal
- Exécuté sans argument, `umask` affiche la valeur actuelle du masque
- Par exemple, on désire retirer au groupe le droit d'écrire et retirer aux autres (*other*) tous les droits :

```
umask 027
```

# Exercice umask

- Observez le umask courant, puis créez un fichier et un répertoire pour vérifier la correspondance
- Changez la valeur du umask, puis créez un fichier pour vérifier la correspondance
- À l'aide d'un test empirique, déterminez si le umask est hérité d'une shell à l'autre

# Configurer bash

---

# Fichiers de configurations

bash utilise plusieurs fichiers cachés à la racine du répertoire *home* pour obtenir une configuration :

<b>.bashrc</b>	Exécuté à chaque nouveau shell (non-login)
----------------	--

---

<b>.bash_profile</b>	Shell de login uniquement ( <i>login shell</i> )
----------------------	--

---

<b>bash_logout</b>	Exécuté lorsqu'on quitte un shell de login
--------------------	--

# Shell de login (.bash\_profile)

- Les shells de login sont ceux pour lesquels **il faut s'authentifier** :
  - connexion à distance
  - console virtuelle (tty)
- Un terminal lancé depuis l'interface graphique n'est pas un shell de login
- La pratique actuelle est d'exécuter<sup>1</sup> `.bashrc` depuis `.bash_profile`

Contenu `.bash_profile` :

```
...  
if [ -f ~/.bashrc ]; then  
    source ~/.bashrc  
fi  
...
```

---

<sup>1</sup>automatique sous Mac

# Configuration du shell (.bashrc)

Habituellement, on utilise `.bashrc` pour définir les éléments suivants :

- Variables d'environnement
- Alias
- Permissions par défaut
- Comportement de bash

# Divers





# Commandes utiles

<code>awk -F":" '{print \$n}'</code>	Affiche la <i>n</i> ème colonne, où le séparateur de colonnes est :
<code>basename FILE</code>	Extrait le nom du fichier sans son chemin
<code>dirname FILE</code>	Extrait le chemin d'un fichier, sans le fichier
<code>sed s/JPEG/jpg/</code>	remplace 'JPEG' par 'jpeg'
<code>uname -a</code>	Affiche les informations du système (nom machine, version noyau, etc.)
<code>cut -c m-n</code>	Affiche les caractères de <i>m</i> à <i>n</i> de chaque ligne
<code>head -n2</code>	Affiche les 2 premières lignes
<code>tail -n2</code>	Affiche les 2 dernières lignes
<code>find pics -name "*.jpg" -exec convert {} {}.png \;</code>	Converti récursivement toutes les images se trouvant dans pics de jpg à png
<code>grep -rn main --include "*.go"</code>	Recherche le <i>pattern</i> main récursivement dans tous les fichiers se terminant par .go et affiche chaque ligne matchée

# Alias

- On peut définir des **alias** pour raccourcir l'invocation de commandes et de leurs arguments
- On peut accéder à la commande originale en utilisant un *backslash* : \
- La commande `alias` exécutée seule affiche tous les alias
- On trouve typiquement ceci dans `.bashrc` :

```
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
alias ls='ls --color=auto'
alias ll='ls -lh'
alias la='ll -a'
```

# Exercice alias

Ajoutez un alias `findn` qui cherche dans le répertoire courant un fichier portant le nom passé en paramètre

# Autres sujets

- Opérations arithmétiques
- Tableaux
- Variables définies par défaut
- Règles pour les guillemets (*quotation*)

- Livre :
  - *Unix Power Tools, 2002, O'Reilly*
- Liens :
  - [Bash Guide for Beginners](#)
  - [Ryan's Bash Scripting Tutorial](#)
  - [Bash Scripting Tutorial: How to Write a Bash Script](#)
  - [Advanced Bash-Scripting Guide](#)