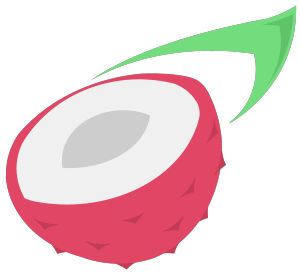

Litchi Pi

Scripting shell

Bash

24 Novembre 2025

Introduction

B ourne **A** gain **Sh** ell

Processeur de commandes, **interface** textuelle avec l'OS

Exécute des commandes dans un *environment*
Peut exécuter des *scripts* de commandes

Bash est simple, mais requiert de connaître beaucoup d'outils CLI (GNU Coreutils)

Il existe des opérations avancées en bash, mais pas obligatoires

Bash suit le standard **POSIX**, son fonctionnement est donc *standardisé*

Cela explique les similarités entre MacOS et Linux

Syntaxe

Les variables bash sont les éléments les plus *bizarres* et **problématiques**

```
# Le signe "=" doit être collé, sinon syntax error
MY_VAR_1="toto"          # Valeur fixe
MY_VAR_2=$(my_command)   # Récupère le résultat de "my_command"
MY_VAR_3=                # Valeur vide, totallement accepté
MY_VAR_4="tutu $MY_VAR_1 ${MY_VAR_2}TOTO"
export MY_VAR_5="titi"   # Valeur disponible pour tout processus créé par le shell courant
```

```
MY_VAR_1="toto"
MY_VAR_2="tutu $MY_VAR_1"      # tutu toto
MY_VAR_3='tutu $MY_VAR_1'      # tutu $MY_VAR_1
```

```
ARGS='tutu toto'
echo "tutu" | grep "$ARGS"    # Tout va bien
#echo "tutu"| grep "tutu toto"

echo "tutu" | grep $ARGS      # grep: toto: No such file or directory
#echo "tutu"| grep tutu toto"
```

```
if my_command; then  # Si la commande "my_command" s'exécute sans erreur
  echo "toto";  # Echo: commande qui affiche un message
fi

if ! my_command; then # Si la commande "my_command" retourne une erreur
  exit 1;  # Exit: commande qui quitte le shell actuel
fi
```

Chaque commande se termine par “;”, sauf si on passe une ligne à la fin

La syntaxe `echo "toto"; if [ -f /tmp/test ]; then echo "tutu"; fi; echo "end"` est valide

```
#Revient à faire:
#if test -f /tmp/test; then
if [ -f /tmp/test ]; then
  echo "File exist"
fi

# Conditions bash permettant des opérateurs plus "classiques"
if [[ "$MY_VAR_1" == "toto" ]]; then
  echo "OK"
fi
```

```
if [ $var -eq 3 ]; then
  echo "Var = 3"
elif [[ $var ==  5 ]]; then
  echo "Var = 5"
else
  echo "Var unknown"
fi
```

```
while <condition>; do
  echo "toto"
done
```

```
for i in 1 2 3 4 5
do
  if [ $i -eq 3 ]; then
    break;
  fi
done
```

```
MY_VAR_1="toto"

my_function() {
  echo "Argument: $1 $2"
  echo "Tous les arguments: $@"
  echo "Nombre d'arguments: $#"
  MY_VAR_1="$2"
  return 3;
}

my_function "toto" "tutu" "titi" "tointoin"
```

Les scripts, et functions ne peuvent retourner qu'un code d'erreur

Tout le reste se fait en écrivant / lisant du texte dans la console

L'environment

La console comporte des **variables d'environment** qui paramétrer son fonctionnement

Les plus importantes sont *automatiquement gérées*, mais vous pouvez tout péter si vous voulez

- **HOME**: Dossier personnel, permet de comprendre les paths relatifs (ex: `/home/juliette`)
- **PWD**: Dossier courant (ex: `/home/juliette/my/folder`)
- **USERNAME**: Votre nom d'utilisateur (ex: `juliette`)
- **PS1**: La ligne qui s'affiche dans l'invite de commande (ex: `\u@\h \w$`)
- **SHELL**: Le shell qui s'exécute actuellement (ex: `bash`)
- **TERM**: Le terminal utilisé pour afficher la console (ex: `xterm-256color`)
- **EDITOR**: L'éditeur de texte par défaut utilisé (ex: `vim`)

Pour éditer l'environment pour le shell courant, et *tous les processus qu'il va créer*

```
export EDITOR="nvim"
```

Commandes

Parmis l'environement, la variable **PATH** permet de trouver les commandes

```
PATH="/bin:/sbin:/usr/bin:/home/juliette/.local/bin
```

Lorsque l'on tape `cd`:

- bash va regarder dans chacun des dossiers de `$PATH`
- S'il trouve un exécutable appelé `cd` dans ce dossier, il l'exécute

Ainsi, il n'y a pas de "commande" bash, juste une **multitude** de petits programmes

Et en créant vos propres scripts et programmes, vous pouvez créer des commandes aussi.

Gestion des erreurs

Chaque binaire exécuté retourne un **nombre**

Si ce nombre est **0**, tout va bien, sinon, une erreur a été rencontrée

```
my_command
echo "$?"  # Affiche le code d'erreur
```

```
if my_command; then
# Pareil que
my_command
if test $? -eq 0; then
```

En terminant le script, on retourne 0 par défaut

Pour retourner une erreur, utiliser `exit <code d'erreur>`

Chaine de commandes:

```
# cmd2 s'exécute si cmd1 ne retourne pas d'erreur
# cmd3 s'exécute si cmd2 (et donc cmd1) ne retournent pas d'erreur
cmd1 && cmd2 && cmd3
```

```
# cmd2 s'exécute si cmd1 retourne une erreur
# cmd3 s'exécute si cmd1 et cmd2 retournent une erreur
# Note: le message d'erreur de cmd1 (et cmd2) s'affichera tout de même
cmd1 || cmd2 || cmd3
```

```
# Si cmd1 réussi, cmd2 est exécuté
# Si cmd1 échoue, cmd3 est exécuté
cmd1 && cmd2 || cmd3
```

Pour interrompre un script en cas d'erreur, on peut changer les options du shell

```
set -e  # Stoppe en cas d'erreur

my_first_command
my_other_command  # Ne vas pas s'exécuter si my_first_command est en erreur
my_final_command  # Ne vas pas s'exécuter si my_other_command est en erreur
```

Note: Beaucoup d'autres options du shell permettent de changer son fonctionnement

Ex: `set -x` permet de voir les commandes exécutées, très utile pour débugger

Flux de données

Examples de flux de données:

```
MY_VAR=$(echo "tutu" | sed 's/u/i/g')

echo "titi" | sed 's/i/u/g' > /tmp/my_file
```

Redirection de **STDOUT** et **STDERR**:

```
my_command 2>&1 > /logfile    # STDERR (2) est redirigé dans STDOUT (1), puis le tout est ajouté à un fichier
echo "tutu" >> /logfile       # On ajoute un message à la fin du fichier (">" l'écraserait)

my_command 2> /error_log 1> /log  # STDOUT (1) dans un fichier, STDERR (2) dans l'autre
```

Pokédex de commandes utiles

- `cat ./mon_fichier`: Lit un fichier
- `echo "Toto"`: Écrit un message dans **STDOUT** (l'affiche)
- `read toto`: Lit **STDIN** (par ex clavier) et stocke le résultat dans la variable `toto`
- `head /my_file`: Affiche le début (ou la fin avec `tail`) d'un fichier
- `sort`: Trie les lignes
- `uniq`: Supprime les doublons de lignes

Note: La plupart de ces commandes fonctionnent en pipe `|` ainsi qu'avec un fichier

Grep

Outil pour extraire des données

```
grep "toto" /my_file      # Récupère toutes les lignes comprenant "toto" dans /my_file
my_command | grep "error" # Récupère toutes les lignes de STDOUT comprenant "error"

cat ./my_file | grep -v "ok" # Récupère toutes les lignes n'ayant PAS le texte "ok"

grep -rn "toto" . # Recherche dans tous les fichers courants le texte "toto"
```

Utilisation avancée avec regexp:

```
cat ./logfile | grep -E "[0-9]{2}$"

cat ./logfile | grep -o -E ".[a-z]{3}$"  # N'extrait que la partie matchée

cat ./logfile | grep -i "toto"    # Case insensitive
```

Awk

Language de script orienté données

En gros: script spécialisé dans l'extraction et formattage de données

```
echo "a b c d e" | awk '{print $3}'    # Affiche "c"
echo "a:b:c:d:e" | awk -F ':' '{print $3 " -> " $1}'  # Affiche "c -> a"
```

Utilisation très avancée méritant un cours à lui tout seul

Sed

Permet de remplacer des données

Utilise des regexp, avec selection, formattage, remplacement.

```
sed "s/harry/henri/g" ./harry_potter.txt    # Sur un fichier

echo "http://toto.example/index.html" | sed "s/toto.example/google.com/g"
# Affiche http://google.com/index.html

echo "http://toto.example/index.html" | sed "s+//+slash-slash+g"
# Affche http:slash-slashtoto.example/index.html

echo "003 33 1" | sed "s/([0-9]{3}) ([0-9]{2}) ([0-9]{1})/\3 \2 \1/g"
# Affiche "1 33 003"
```

TP Time

Analyse de logfile avec bash