

Menu du jour :

- recap du dernier CM et TD
- installation et mise-à-jour d'un système Linux basé sur Debian
- programmation en shell (bash, pwsh, cmd)



Recap du dernier CM et TD

- La différence entre un programme et un processus
- La mythologie des processus
- Les successions de pipes
- Exécuter vs sourcer vs remplacer
- Les commandes internes et externes

Programme, processus, commutation : métaphore

Une informaticienne prépare un gâteau d'anniversaire pour sa fille. Elle a une recette pour faire le gâteau et dispose de farine, d'oeufs, de sucre ...



1

Allez sur wooclap.com

2

Entrez le code d'événement dans le bandeau supérieur

Code d'événement
MYKUYR



Activer les réponses par SMS

Programme, processus, commutation : métaphore

Une **informaticienne** prépare un gâteau d'anniversaire pour sa fille. Elle a une recette pour faire le gâteau et dispose de farine, d'oeufs, de sucre ...

- La **recette** représente le **programme** (algorithme traduit en une suite d'instructions).
- L'**informaticienne** joue le rôle du **processeur** (CPU)
- Les **ingrédients** sont les **données** à fournir
- Le **processus** est l'**activité** de notre cordon bleu qui lit la recette, trouve les ingrédients nécessaires et fait cuire le gâteau.

Si le fils de l'informaticienne arrive en pleurant parce qu'il a été piqué par une guêpe, sa mère marque l'endroit où elle en était dans la recette (l'état du processus en cours est sauvegardé), cherche un **livre sur les premiers soins** et commence à **soigner son fils**.

Le **processeur** passe donc d'un processus (la **cuisine**) à un autre plus prioritaire (les **soins médicaux**), chacun d'eux ayant un **programme** propre (la **recette** et le **livre des soins**).

Lorsque la piqûre de la guêpe aura été soignée, l'**informaticienne** reprendra sa **recette** à l'endroit où elle l'avait abandonnée.



Recap : la mythologie des processus



1

Allez sur wooclap.com




2

Entrez le code d'événement dans le bandeau supérieur

Code d'événement
MYKUYR

 Activer les réponses par SMS

Recap : la mythologie des processus

-  **Zombie** Achevé, défunt. Doit être traité par son père.
-  **Orphelin** A perdu son père, va être adopté par **init**
-  **Démon** N'est pas crée par l'utilisateur

Communication entre processus

Suite de tubes pour le filtrage des données :

```
prompt> df -k . | tail -1 | sed "s/ */ /g" | cut -d " " -f 4  
60833156
```



1

Allez sur wooclap.com

2

Entrez le code d'événement dans le bandeau supérieur

Code d'événement
MYKUYR

 Activer les réponses par SMS

Communication entre processus

- Affichage des statistiques en kilo-octet (option **-k**) sur le répertoire courant (`.`) :

```
prompt> df -k .
```

```
Filesystem 1K-blocks Used Available Use% Mounted on  
/dev/sda7 98123404 32281532 60833156 35% /
```

- On ne garde qu'une seule ligne en partant de la fin :

```
prompt> df -k . | tail -1
```

```
/dev/sda7 98123404 32281532 60833156 35% /
```

- On ne garde qu'un seul espace entre chaque mot :

```
prompt> df -k . | tail -1 | sed "s/ */ /g"
```

```
/dev/sda7 98123404 32281532 60833156 35% /
```

- On ne garde que le quatrième champ de la ligne, l'option « **-d** » précise le séparateur à prendre en compte (l'espace) :

```
prompt> df -k . | tail -1 | sed "s/ */ /g" | cut -d " " -f 4
```

```
60833156
```

⇒ La commande affiche l'espace libre (en kilo-octet) sur la partition qui contient le répertoire de travail.



Recap : sourcer vs exécuter vs remplacer



1

Allez sur wooclap.com

2

Entrez le code d'événement dans le bandeau supérieur

Code d'événement
MYKUYR

 Activer les réponses par SMS



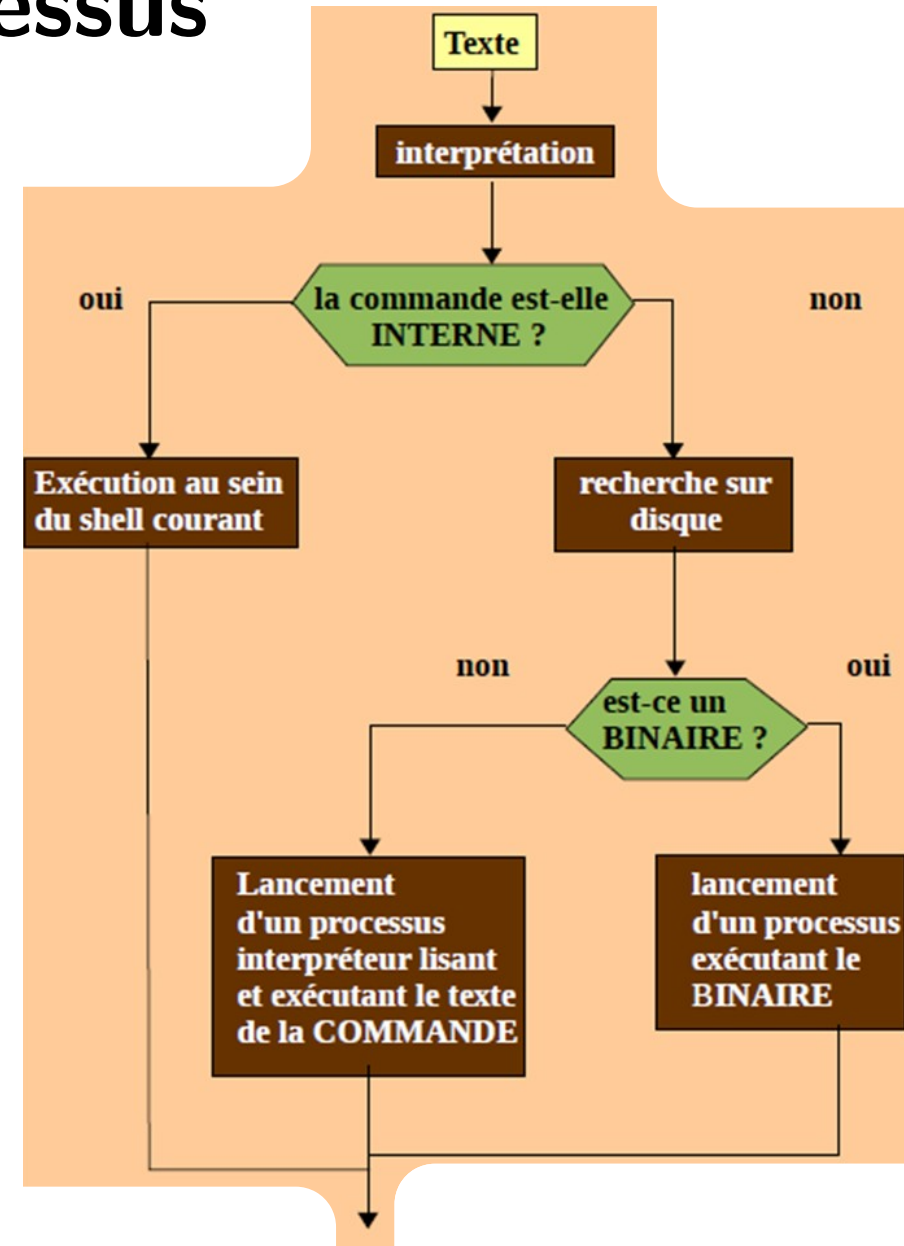
Recap : sourcer vs exécuter vs remplacer

- Sourcer : ne crée pas de nouveau processus (**source f** ; **. f**)
- Exécuter : crée un nouveau processus (**./f**, **chemin/vers/f**, **f** – si dans le PATH)
- Exec : remplace le processus courant → on ne finira pas les instruction du « père »

Création ou non de nv processus

Nouveau processus

Oui	Non
<code>./f</code> <code>bash f</code> <code>f</code> Commandes externes	<code>fource f,</code> <code>. f</code> <code>exec f</code> (remplace le processus courant) Commandes internes





Linux : les infos systèmes et la mise à jour

Info système : versions Linux

Coté noyaux :

- convention de numérotation **x.y.z** :
 - **x** : numéro de version.
 - **y** : si pair, désigne une version stable, sinon, désigne une version en Bêta-test.
 - **z** : incrémenté à chaque correction de bug.
- Pour connaître la version du noyau en cours :

```
prompt> uname -r  
5.15.0-47-generic
```

Coté système :

Pour connaître la distribution utilisée :

```
prompt> cat /etc/issue  
Ubuntu 22.04.1 LTS
```



La commande `uname`

Affiche les informations relatives à la version du système. L'option `-a` (all) affiche toutes les informations.

Exemple :

```
nanis@jammy:~$ uname -a  
Linux C302L-G24P07.png.unicaen.fr 6.8.0-45-generic #45~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC  
Wed Sep 11 15:25:05 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

<https://www.geeksforgeeks.org/uname-command-in-linux-with-examples/>



Mise à jour du système Linux

- Il faut **mettre régulièrement son système linux à jour**
 - pour corriger des bugs,
 - ajouter des logiciels,
 - supprimer les logiciels obsolètes, ...
- Vocabulaire :
 - Un **paquet** (package) est un fichier qui permet d'installer un logiciel sur une distribution. Le fichier exécute des scripts afin de placer les fichiers (de configuration et de l'application) au bon endroit sur le système.
 - Les paquets sont créés et maintenus par des **mainteneurs**
 - Les paquets sont (souvent) stockés sur des serveurs, appelés **dépôts (repository)**.
dépôts officiels vs dépôts tiers (third-party)
 - Un **gestionnaire de paquets** est un outil qui permet de gérer des logiciels d'une distribution depuis les dépôts.



Mise-à-jour sur les distributions Debian

Distribution tq Ubuntu, Mint, ...

Gestionnaire de paquets : **Advanced Package Tool (APT)**

Paquet : fichiers **.deb**.

- **Installation/ mise-à-jour d'un logiciel :**
 - APT se connecte à l'un repository
 - APT télécharge le .deb
 - APT installe le .deb
 - De manière sous-jacente : commande **dpkg**.
- **Mise à jour de la distribution :**
 - Télécharger tous les .deb de la nouvelle version et les installer.

<https://www.malekal.com/apt-installer-mise-a-jour-paquet-distribution-debian-ubuntu-mint/>



APT : fichiers de configuration

- `/etc/apt/sources.list` : stocke les sources avec l'adresse des dépôts
- `/etc/apt/sources.list.d/` : sources additionnels. Ainsi on peut ajouter des dépôts non officielles.
- `/etc/apt/apt.conf` : fichier de configuration APT
- `/etc/apt/apt.conf.d/` : fichiers de configuration additionnels.
- `/etc/apt/preferences.d/` : Les fichiers de préférences additionnels.
- `/var/cache/apt/archives/` : Stocke les les deb déjà téléchargés (évite de retélécharger si réinstallation)
- `/var/lib/apt/lists/` : stocke la liste et informations sur les packages du systèmes.

Les commandes APT

Apt regroupe différentes commandes, selon les besoins.

On modifie la configuration système → ces opérations **nécessite des droits root** :

- s'identifier en root (**su**)
- utiliser **sudo**

dpkg : le programme qui installe les fichiers .deb

apt-* (**install/cache/key**): programmes qui traquent les paquets disponibles, les téléchargent et les filent à **dpkg**

apt : wrapper pour **apt-*** ← **partez du principe que c'est apt que vous aller utiliser**

aptitude : un frontend pour APT



La commande `sudo apt update`

- Met à jour (resynchronise) l'**indexation** du dépôt sur **vosre** Linux.
- En effet, indexation trop ancienne = plus synchronisée avec l'indexation en ligne. Ainsi, vous pouvez demander une version qui n'existe plus et obtenir une erreur.
- Enfin on utilise `apt update` lorsque l'on modifie le sources afin de télécharger les nouveaux index.



La commande `sudo apt upgrade`

Met à jour **les paquets** de la distribution Linux de votre machine (pas la version de la distro en elle même...).

En effet, des mises à jour de sécurité sont publiées chaque jour.

Lancement de la commande, **affichage** de la liste des mises à jour (potentiellement très longue si la dernière mise à jour remonte à très longtemps), **validation** de l'utilisateur (**o/y**), **téléchargement** des paquets (la vitesse et le délai s'affichent en bas à droite de l'écran), phase **d'installation**.

`apt` peut poser des questions sur des actions à effectuer durant la mise à jour.



Autre commandes `apt`

Pour supprimer les fichiers qui ne sont anciens ou plus nécessaires.

- **`sudo apt autoremove`**

supprime les paquets installés dans le but de satisfaire les dépendances d'autres paquets et qui ne sont plus nécessaires.

- **`sudo apt clean`**

vide le dossier `/var/cache/apt/archives` qui contient les `.deb` téléchargés.

- **`sudo apt autoclean`**

nettoie le référentiel local des paquets récupérés. La différence avec `clean` est qu'il supprime uniquement les paquets qui ne peuvent plus être téléchargés et qui sont inutiles.



La commande `sudo apt dist_upgrade`

- Met la **distribution à niveau** (fait passez à la version suivante de votre distribution).
- Ensuite on lance **apt upgrade** ou encore **apt-full upgrade** pour finir la mise à jour,



Les scripts shell

Définitions et concepts généraux

- **Script** = série de commandes dans un fichiers.
- **Langage interprété** : exécute les instruction directement, sans avoir besoin de compilation. Les langages shell (sh, **bash**, ksh, zsh, **pwsh**) sont interprétés (comme aussi python, mais pas comme C).
- Les **extensions** sont des *conventions* (contrairement à Windows, ce ne sont pas les extension qui détermine le type de fichiers...).
- **Shebang** : première ligne d'un script, indiquant l'interprète à utiliser « shebang »
`#!/bin/bash` `#!/usr/bin/env pwsh` `#!/usr/bin/env python3`
- **Code de retour** : *par convention*, 0 signifie que tout s'est bien passé. Les valeurs supérieures à 0 représentent différents cas d'erreurs (sémantique à documenter)
- **Syntaxe** : syntaxe variant en fonction du shell (→ utiliser des **cheatsheets**), mais un script qui respecte la norme POSIX est en principe compréhensible par n'importe quel shell...



Éléments de langages

- Les variables.
Un contenant nommé ayant une valeur. Doit être **déclarée avant d'être utilisée** (affichage, calcul, ...)
Type de variable : numérique, chaîne, tableau, ...
- Les structures conditionnelles. **Si/Sinon/Si**
- Les structures itératives.
 - **Pour**
 - **Tant que**
- Les fonctions.

Exécution d'un script shell

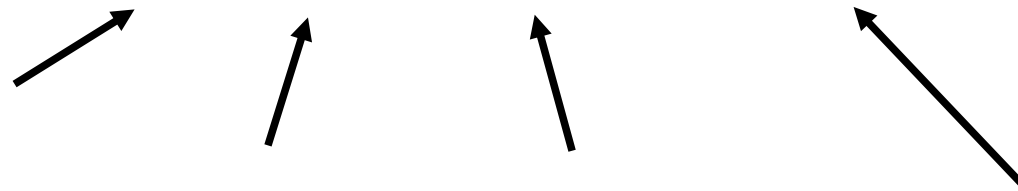
Le shell suit grosso modo les étapes suivantes ; il :

- Lit le script **ligne à ligne**, et **charge l'interpréteur** spécifié par le **shebang**.
- Coupe la ligne en morceaux (**tokens**) et se demande :
 - Quelle est la **commande** ? (fonction / commande intégrée (built-in) / exécutable / script)
 - Quels sont les arguments / options / paramètres ?
- Exécute les **expansions** (***.c** transformé en liste de chemins, par exemple).
- Exécute les actions de **redirection** (**>toto** ouvre en lecture le fichier **toto**)
- Exécute la commande ; les arguments sont numérotés de 1 à n.
- Attend que la commande soit finie et récupère le statut en sortie.

Les scripts bash

- Extension **.sh** par *convention* et shebang : **#!/bin/bash**
- Langage respectant la **norme POSIX**
- Exécution : **bash file**, **/chemin/vers/le/file**, **./file**, **file**, **. file**, **source file**
/!\ à la **syntaxe**, aux **droits** et au **PATH** (cf TD)
- Code de retour : max 255, par défaut : 0, mots clés **return** ou **exit**
- **Valider ses scripts** pour éviter les erreurs courantes et mauvaises pratiques :

<https://www.shellcheck.net/>





Script bash : les variables

Script bash : les variables chaîne de caractères

- Déclaration, via le `=`, ou le `(())` :

une variable appelée « `s` » qui contient un « `toto` » → type chaîne

```
s='toto' # /!\ pas d'espace !!!!!!!!!!!!!
```

```
((s = "toto" ))
```

- Utilisation en affichage :

```
echo s # affiche « s »
```

```
echo $s # affiche le contenu de s, donc « toto »
```

```
echo "M. $s" # utiliser " et pas ' pour que la variable soit interprétée
```

```
echo "M. \"$s\"" # /!\ Il faut échapper les guillemets si nécessaire
```

```
echo "tototo${s}tototo" # /!\ Il faut délimiter la variable si nécessaire
```

Script bash : les variables numériques

Déclaration, via le `=`, ou le `(())`, (ou le `let`) :

une variable appelée « `a` » qui contient un `1` → type numérique

```
$ v=1 # /!\ pas d'espace !!!!!!!!!!!!!!!
```

```
$ (( v = 1 ))
```

```
$ let "v=1" # préférer la syntaxe (( )) que le let, voir shellcheck ;)
```

Utilisation en affichage :

```
$ echo $v # affiche (echo) le contenu de la variable v
```

```
$ echo "Taille du disque : ${v}To"
```

```
$ echo "blablabla $(( v ))"
```

Script bash : les variables numériques

Utilisation en calcul :

- Bash ne peut pas faire directement de calculs mathématiques :
`$ a=2 ; echo $a+2`
`$a+2`
- Utiliser `$(())` (ou les mot-clé `let` et `expr`) pour des opérations sur les entiers
Opération prises en charge : `+`, `-`, `*`, `/`, `**` (puissance), `%` (modulo)
`$ a=$((2 + 2)) ; echo $a # 4, /!\ pas d'espace pour l'assignement`
`$ b=$((a / 3)) ; echo $b # 3 /!\ nombres entiers...`
- Contraction d'opérations (à l'instar de nombreux langages de prog) :
`$ a=1 ; ((a+=1)) ; echo $a # affiche 2`
`$ a=1 ; echo $((a+=1)) # on utilise le $ pour récupérer l'output directement`
- Autres possibilités pour effectuer des calculs (pas forcément avec les nombres entiers) : `bc`.

https://fr.wikibooks.org/wiki/Programmation_Bash/Calculs, <https://github.com/koalaman/shellcheck/wiki/SC2219>

Script bash : les variables tableaux

- **Déclaration :**

```
t=("un et " "deux et " "trois et " "quatre")
```

- **Utilisation :**

```
echo ${t} # affiche juste le premier élément...
```

```
echo ${t[@]} # affiche tout le tableau
```

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

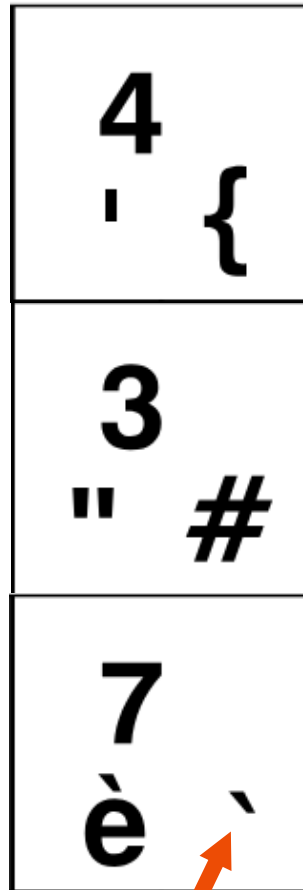
```
echo ${t[1]}
```

- **Redéfinition :**

```
t[1]="toto"
```


Script bash : les trois types de quotes

- **Simple quote** « ' ' » : contenu pas analysé et traité de façon brute
`$ echo 'je m\'appelle $prenom.'`
Je m'appelle \${prenom}.
- **Double quotes** « " " » : le contenu est analysé et traité (pour \$, \ et `).
`$ echo "je m'appelle $prenom."`
Je m'appelle Athénaïs
- **Back quotes** « ` ` » : utilisation du résultat (sortie standard) de l'exécution de la commande entre backquotes
Alternative : `$()` (à préférer aux « ` ` », en fait...)
`$ echo "Vous êtes sur `uname`"`
`$ echo "Vous êtes sur $(uname)"`



AltGr



Script bash : les arguments

Quand on lance un script avec des arguments *scriptname arg1 arg2 arg3*, le shell fait les assignments de **variables spéciales** : **$\$1=arg1$** , **$\$2=arg2$** , **$\$3=arg3$**

Aussi :

- **$\$0=scriptname$**
- **$\$#$** : nombre d'arguments du script (sans compter **$\$0$**)
- **$\$@$** : liste des arguments du script
- **$\$?$** : code de retour de la dernière commande



Script bash : les arguments

Exemple :

--- dans un script :

```
if test "$#" -eq "0" then
    echo "usage: $0 arg1 arg2" >&2 exit 1
fi
```

→ Si le nombre d'arguments passés à l'appel de la commande vaut 0, on quitte la commande avec un message d'erreur sur la sortie standard (sortie 1 copiée sur la sortie 2).

Script bash : saisie de l'utilisateur.ice

La commande **read** permet de récupérer la saisie de l'utilisateur.ice

```
$ read nom # Demande à l'utilisateur.ice de saisir une valeur  
$ echo "Bonjour ${nom} !"
```

```
$ read nom prenom # Saisir plusieurs variables d'affilée  
$ echo "Bonjour $prenom $nom !"
```

read lit **mot par mot** (le séparateur est l'espace). Chaque mot est affecté aux variables spécifiées dans l'ordre où elles sont données. La dernière variable récupère tous les mots restants s'il y en a plus que spécifié.

Quelques options :

- **-p** : Affiche en plus un message pour l'utilisateur.

```
read -p 'Entrez votre nom : ' nom  
echo "Bonjour $nom !"
```
- **-s** : Masque le texte saisi (pratique pour un mot de passe, par exemple)
- **-t s** : Renvoie une valeur vide dans la variable au bout de **s** secondes.

```
read -t 15 -p 'Entrez votre nom dans les 15 secondes qui suivent : ' nom
```



Script bash : les tests

Les tests : syntaxes POSIX

Syntaxes :

- Avec la commande « **test condition** » : `test "${login}" = "toto"`
- Avec les « **[condition]** » : `["${login}" = "toto"]` /!\ « **[]** »

sur nombres

N1 -eq N2 : Vrai si les nombres sont égaux (equal, **==**)

N1 -ne N2 : Vrai si les nombres sont différents (not equal, **!=**)

N1 -lt N2 (less than, **<**)

N1 -le N2 (less equal, **<=**)

N1 -gt N2 (greater than, **>**)

N1 -ge N2 : (greater equal , **>=**)

sur fichiers

-e FICHIER : Le fichier existe

-f FICHIER : C'est un fichier ordinaire

-d FICHIER : C'est un répertoire

-L FICHIER : C'est un lien symbolique

-r FICHIER : Le fichier est lisible (read)

-w FICHIER : Fichier modifiable (write)

-x FICHIER : Fichier exécutable (execute)

FICHIER1 -nt FICHIER2 : F1 plus récent que F2

FICHIER1 -ot FICHIER2 : F1 plus ancien que F2

Retour :

- **0** si le test est vrai,
- **1** si le test est faux,
- **2 ou plus** si erreur.

sur chaînes de caractère :

=, !=, <, etc.

-z CHAINE : Vrai si la chaîne est vide

-n CHAINE : Vrai si la chaîne n'est pas vide

Exemple : `["${nom}" = "prenom"]`



Les tests : exemples simples

```
prompt> ls -l
drwxr-x--- 11 c1 cours 17 Aug 1 09:00 save
-rw-r----- 1 fun axis 21 Jul 25 17:05 data
```

```
prompt> who am i
c1 term/c4 Aug 2 09:01
```

```
prompt> test -f save; echo $?
1
```

```
prompt> test -d save; echo $?
0
```

```
prompt> test -r save; echo $?
0
```

```
prompt> test -f data; echo $?
0
```

```
prompt> test -w data; echo $?
1
```

Les tests : combinaisons

Avec `test` ou en dehors des `[]`:

- « Et » logique : `&&`
`test EXPR1 && test EXPR2`
`[EXPR1 && EXP2]`
- « Ou » logique : `||`
`test EXPR1 || test EXPR2`
`[EXPR1 || EXPR2]`

Directement à l'intérieur des `[]`:

- « Et » logique : `-a` : `[EXPR1 -a EXPR2]`
- « Ou » logique : `-o` : `[EXPR1 -o EXPR2]`
- Inverser un test : `!` : `[! EXPR]`

Combinaisons avancées :

- avec `{ }`
`if ["true"] || { [-e /does/not/exist] && [-e /does/not/exist] ;} ; then echo true; else echo false; fi`
- avec `()` : il faut les échapper ou les mettre entre `' '` → pas très lisible !
`if [\("true" -o -e /does/not/exist \) -a -e /does/not/exist]; then echo true; else echo false; fi`
`if ['(' "true" -o -e /does/not/exist ')' -a -e /does/not/exist]; then echo true; else echo false; fi`



Les tests : syntaxes alternatives

`[condition]` et `test condition` sont équivalents (et **POSIX**, donc portable ! :D)

`[[condition]]` : amélioration de `[condition]`. Dispo en ksh, bash, zsh...

`((condition))` : utilise l'expansion arithmétique. Dispo en ksh, bash, zsh.

Discussion sur les différences entre `[]` et `[[]]` : <https://forum.ubuntu-fr.org/viewtopic.php?id=398332>

<https://unix.stackexchange.com/questions/306111/what-is-the-difference-between-the-bash-operators-vs-vs-vs>



Script bash : les structures de contrôle

- Les conditions
- Les boucles
 - Tant que
 - Pour



Les conditions

si / **si-sinon** / **si - sinon si - [sinon si ...] - sinon**

if *condition* ; **then**

instructions

elif *condition* **then** # autant de elif qu'on veut

instructions

else # max un else par if, et en dernier !

instructions

fi



Les conditions : exemple de si-sinon

```
----  
if test -d "$1" -a -x "$1" then  
    echo chemin accessible  
    cd $1  
else  
    echo chemin inaccessible  
fi
```

→ **Si** la valeur du premier argument est un répertoire **et** que l'on est autorisé à se déplacer dedans (**-x**), **alors** on y va. **Sinon** on affiche un message.

Les conditions : un cas pratique

Consigne : Vérifier si une variable est un nombre :

```
[ $mvariable -eq 1 ] 2> /dev/null
if [ $? -eq 0 -o $? -eq 1 ]
then
    echo "C'est un nombre."
else
    echo "Ce n'est pas un nombre."
fi
```

Explications :

`[$mvariable -eq 1] 2> /dev/null`

Compare la valeur de la variable au nombre « 1 ». Trois cas pour le code de retour du test :

- 0 : La valeur de la variable est égale à 1, donc elle vaut 1.
- 1 : La valeur de la variable est différente de 1, mais la comparaison s'est bien réalisée → la valeur de la variable est un nombre != 1
- 2 ou + : La comparaison a échoué → la valeur de la variable n'est pas un nombre.

```
if [ $? -eq 0 -o $? -eq 1 ]
```

si le code de retour est 0 ou 1, on a un nombre.

sinon, ce n'est pas un nombre.

/!\ Il faut impérativement utiliser le « -o » et non « || ».

`if [$? -eq 0] || [$? -eq 1]` ne fonctionne pas car on effectue deux tests différents :

- Au premier test, `$?` contient le code de retour du test de la variable,
- Au 2nd test, `$?` contient le code de retour du 1er test du « if » actuel. On ne testerait donc pas le bon code retour.

Conditions et combinaison de commandes - ET

Exécuter *cmd2* uniquement si la commande *cmd1* se termine correctement : *cmd1 && cmd2*

Exemple : S'il existe un répertoire *tmp* dans le répertoire courant, alors aller dans ce répertoire.

```
$ pwd
/home/c1
$ mkdir tmp
$ test -d $HOME/tmp && cd $HOME/tmp
$ pwd
/home/c1/tmp
```

```
$ cd
$ rmdir tmp
$ test -d $HOME/tmp && cd $HOME/tmp
$ pwd
/home/c1
```

En utilisant la structure de contrôle **if ... then ...fi** :

```
$ pwd
/home/c1
$ mkdir tmp
$ if test -d $HOME/tmp
> then cd tmp
> fi
$ pwd
/home/c1/tmp
```

```
$ cd
$ rmdir tmp
$ if test -d $HOME/tmp
> then cd tmp
> fi
$ pwd
/home/c1
```

Conditions et combinaison de commandes - OU

Exécuter *cmd2* uniquement si la commande *cmd1* ne se termine correctement : *cmd1* || *cmd2*

Exemple : S'il n'existe pas de répertoire *tmp* dans le répertoire courant, alors afficher un message.

```
$ pwd
/home/c1
$ mkdir tmp
$ test -d $HOME/tmp || echo $HOME/tmp inexistant

$ rmdir tmp
$ test -d $HOME/tmp || echo $HOME/tmp inexistant
/home/c1/tmp inexistant
```

En utilisant la structure de contrôle **if ... then ...fi** :

```
$ pwd
/usr/c1
$ mkdir tmp
$ if test ! -d $HOME/tmp
> then
>     echo $HOME/tmp inexistant
> fi

$ rmdir tmp
$ if test ! -d $HOME/tmp
> then
>     echo $HOME/tmp inexistant
> fi
/usr/c1/tmp inexistant
```

Les conditions : enchainements de cas avec **case**

```
case $nom in
  "Athénaïs") # un cas se termine par )
    echo "Salut Athénaïs !" # Tout le code s'exécute jusqu'au prochain ;;
  ;;
  "Jean-François")
    echo "Bonjour JFA"
  ;;
  M*) # utilisation, de jokers. Ici : tous les noms commençants par « M » ; on fait rien
  ;;
  "Riri" | "Fifi" | "Loulou") # on combine plusieurs valeurs
    echo "Bonjour neuveu"
  ;;
  *) # cas par défaut, si aucun autre test ne valide la valeur de la variable.
    echo "Hein, mais t'es qui ?"
  ;;
esac
```




Les boucles tant que

Syntaxe :

```
while condition ; do  
    instructions  
done
```

Exemple :

```
#!/bin/bash  
while [ -z "$reponse" ] || [ "$reponse" != 'oui' ]  
do  
    read -p 'Dites oui : ' reponse  
done
```

Les conditions sont les mêmes que pour les if

Les boucles for

Syntaxe générale :

```
for name [ in [word ... ] ]  
do  
    instruction  
done
```

```
for v in 'var1' 'val2' 'val3' ; do  
    echo $v  
done
```

```
for i in $@ ; do  
    echo $i  
done
```

```
for (( i=0 ; i<10 ; ++i )) ; do  
    echo $i  
done
```

Pour parcourir le résultat d'une commande :

```
liste_fichiers=`ls`  
for fichier in $liste_fichiers ; do  
    echo "Fichier trouvé : $fichier"  
done
```

Pour renommer les fichiers d'un répertoire :

```
for fichier in `ls` ; do  
    mv $fichier $fichier-old  
Done
```

La commande `seq x y` génère tous les nombres allant de `x` à `y`

```
for i in $(seq 1 10); do  
    echo "n°$i"  
done
```

Alternative (à partir de la version 3 de bash) : `{x..y}`

```
for i in {1..15}; do  
    echo "n°$i"  
done
```

Les fonctions

Déclaration AVANT le premier appel :

```
nomfonction() # les paramètre ne sont pas déclaré
{
    instructions
    return x # ou exit (facultatif, par défaut 0)
}
```

Appel :

```
nomfonction param_1 param_2 ... param_n
```

Remarques :

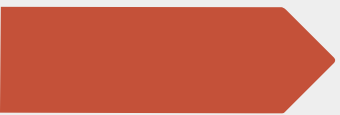
Les fonctions sont similaires aux scripts.

- Les paramètres sont facultatifs.
- Accès au nombre de paramètres transmis via \$#
- Accès au paramètre transmis via \$0, \$1, \$2, ...
- \$0 est le nom de la fonction

Exemples :

```
show() {
    echo 'Hello $1'
    return 5
}
show toto # Hello toto
```

```
isFruitRouge(){
if [ "$1" == "fraise"
    -o "$1" == "framboise"
    -o "$1" == "groseille" ]
then
    return 0 # Exécution correcte (ou vrai, en l'occurrence)
else
    return 1 # Exécution incorrecte (ou faux, en l'occurrence)
fi
}
```



Powershell et cmd



Powershell (et CMD)

CMD pour DOS (l'ancêtre de Windows NT)

Extension : .bat ou .cmd

Powershell : le successeur, depuis 2006. **Orienté objet** (comme python et javascript)

Extension : .ps1

Commandes de la forme **prefixe-objet**

- exemples de préfixes : **get, set, add, clear, import, export, new, write**
- exemples d'objet : **command, item, content, ...**

Compatible POSIX → **cd, ls, help, mkdir**, les redirections, les pipes

Tourne sur Windows (nativement...) mais aussi sur Linux ! ;)

<https://learn.microsoft.com/fr-fr/powershell/scripting/overview?view=powershell-7.4>

<https://simvil.github.io/files/noSQL/CM-4-powershell.pdf>



TD 7 et 8 :

programmation en shell

4 TP :

- shell (bash)
- windows CMD x2
- powershell

