SCRIPTING SHELL

Sommaire

Le She bang

A chaque début de script shell, il est nécessaire d'ajouter le shebang `#!` suivi du shell choisi.

Afin d'éviter des erreurs, il est conseillé de spécifier le Shell à utiliser grâce au she bang.

- Il est matérialisé au moyen de la chaîne #! suivie du chemin du Shell à utiliser (exemple : #!/bin/bash).
- Il est possible d'indiquer le SH, le KSH, le Python, le Perl, ou tout autre langage interprété.

- S'assurer du chemin spécifié, en utilisant par exemple la commande `which`

```
#!/bin/bash
```

Les Commentaires

L'ajout de commentaires dans un script permet d'améliorer sa lisibilité par une tierce personne ou par nous-même ainsi que de commenter les modifications apporter. Ces derniers sont précédés par un `#`

```
#!/bin/bash
# ce script affiche la phrase « Hello World ! »
echo Hello world ! # on peut commenter après une instruction
```

Notez que le #Shebang n'est pas un commentaire.

- L'affectation d'un entête de script complet fait partie des bonnes pratiques.

```
#!/bin/bash
#=========================================================
#
#
# Entête de script - version: 1.0
#
#
#=========================================================
```

Fichier Executable

Pour qu'un script soit exécutable il faut que ce dernier possède la permission *`execute`*

```
chmod +x script.sh
```

Debuguer

Les erreurs dans les scripts ne sont pas toujours faciles à repérer. Plusieurs méthodes peuvent être utilisées afin de les identifier:

- L'exécution du script en mode trace permet d'exécuter une à une les commandes de celui-ci tout en les affichant précédées du signal plus (+).

```
bash -x monscript.sh
+ echo Hello world
Hello world
```

- L'intégration dans le script de commandes de débogue se traduit par l'ajout temporaire de commandes permettant d'analyser une situation. Cela permet de contrôler le résultat d'exécution de certaines parties du script et d'identifier des sources potentielles de dysfonctionnement.

```
echo "tapez votre nom"
read nom
## echo de debug avec pause :
echo "valeur de la variable nom : $nom"
read # pause avant de poursuivre le déroulement du script
## fin du debug
echo "Bienvenue $nom sur la machine $HOSTNAME"
```

Les lignes de debug doivent être commentées avant la mise en production:

```
echo "tapez votre nom"
read nom
## echo de debug avec pause :
# echo "valeur de la variable nom : $nom"
# read # pause avant de poursuivre le déroulement du script
## fin du debug
echo "Bienvenue $nom sur la machine $HOSTNAME"
```

Commandes utiles en Script

`echo` permet d'afficher du texte.

```
echo [option] "expression"
```

Si "expression" contient des caractères spéciaux, il faut la protéger avec des doubles quotes et lui ajouter l'option –e. On peut de plus utiliser des options de mise en forme:

- `\c` inhibe le changement de ligne en fin de commande
- `\n` nouvelle ligne
- `\t` tabulation horizontale
- `\\` le caractère \

`clear` Commande simple sans argument, permettant d'afficher une page écran vierge.

```
exit [ n ]
```

`n` est la valeur retournée au Shell père dans la variable `$?` Un script est généralement exécuté dans un environnement enfant. Celui-ci peut se terminer par la commande « `exit 0` » ou non (dans ce cas, celle-ci est implicite).

Lorsqu'une commande interne est spécifiée en argument, affiche l'aide de la commande.

- Afficher les commandes internes au Shell `help`
- Afficher l'aide de la commande interne `wait`: `help wait`

Les Variables

Les variables sont des éléments essentiels à la réalisation de scripts. Elles permettent de stocker temporairement des informations au cours du déroulement du script Ce contenu peut être affiché ou réutilisé par d'autres commandes.

Les variables peuvent être classées en 4 catégories.

Les variables locales	**Les variables d'environnement**	**Les variables réservées**	**Les constantes (variables en lecture seules)**
Les noms doivent être en minuscule (utilisation de doubles quotes " pour séparer les mots.)	Variables systèmes comme HOME (ou variables exportées)	Il est impossible d'utiliser leurs noms pour des variables locales ou d'environnement	Variable en lecture seule via `declare -r <var>`. Leurs noms doivent être en majuscule (utilisation de "_" pour séparer les mots)

- Une variable locale est utilisable dans l'environnement courant.
- Pour affecter un contenu à une variable : **nom=valeur**
- Pour utiliser le contenu d'une variable, on préfixe son nom par le caractère dollar `$` , généralement entouré de doubles quotes. Utilisation de variables locales

```
errormsg="une erreur est survenue, contactez votre
administrateur système"
logdir="/var/log/"
```

```
[...]
echo "$errormsg"
mkdir -p "$logdir" 2>/dev/null
```

- Les variables réservées

Ce sont des variables dont le nom est réservé par le Shell qui en gère le contenu. Elles sont très utilisées dans les scripts et les enchaînements de commandes.

Variables	**Designations**

Variables	Designations
`$$`	Numéro du processus en cours
`$?`	Code retour (0 si vrai, different de 0 si faux) exit 2 -> provoque la sortie avec le code retour “2”
`$!`	Numéro du dernier processus lancé en arrière-plan
`$#`	Nombre de paramètres reçus par le script
`$1` à `$9`	Paramètres de la commande (1 pour le premier, 2 pour le deuxième, etc.)
`$0`	Nom de la commande ou du script
`$@`	ou $* Valeurs des paramètres reçus encadrés par “ “ . Certaines versions Unix concatènent les valeurs avec *

Exemple:

./monscript.sh valun valdeux valtrois `$#`

`$#` = 3 Trois paramètres reçus

`$0` = ./monscript.sh

`$1` = valun

`$2` = valdeux

`$3` = valtrois

`$@` = valun valdeux valtrois

Modification du contenu d’une variable d’environnement

Une variable d’environnement est une variable qui est héritée dans tous les sous-Shell. Pour créer une variable d’environnement, on peut :

- soit exporter une variable locale

```
export NOMVAR
```

- soit créer (ou modifier) et exporter directement une nouvelle variable

```
export NOMVAR=valeur
```

La commande set

La commande `set` permet notamment d’affecter des valeurs aux variables réservées 1, 2 ...

```
set "$LOGNAME" $(uname -n)
echo "$1 $2"
user20 serveur102
```

Si aucun argument n'est donné, toutes les variables du Shell sont affichées.

La commande read

La commande read est utilisée pour affecter un contenu saisi par l'utilisateur à une variable.

```
read [option] <variable(s)>
```

Cette commande génère l'attente de saisie clavier validée (par Entrée) et procède au stockage de cette saisie dans la (ou les) variable(s) spécifiée(s).

```
echo "Veuillez entrer votre nom : "
read nom
echo "Bonjour $nom"
```

Il est possible de substituer la commande echo par l'option `-p` de `read`.

```
read -p "Veuillez entrer votre nom : " nom
```

Si plusieurs paramètres séparés par des espaces sont fournis à `read`, alors chaque mot saisi est affecté à la variable correspondante. Les saisies en surnombre sont affectées au dernier nom de variable indiqué.

```
read -p " Saisissez votre nom et prénom : " nom prenom
Dupont Jean dit toto
echo "Nom saisi : $nom "
Nom saisi : Dupont
echo "Prénom saisi : $prenom "
Prénom saisi : Jean dit toto
```

Pour éviter ce type de désagrément, il est possible de déclarer une variable tampon non utilisée.

```
read -p " Saisissez votre nom et prénom : " nom prenom tampon
Dupont Jean dit toto
[…]
echo "Prénom saisi : $prenom "
Prénom saisi : Jean
```

Il est éventuellement possible ultérieurement de détruire cette variable tampon

```
unset tampon
```

Les Enchaînements de commandes

Inconditionnel

Pour enchaîner des commandes quel que soit le résultat de leur exécution, on utilise le `;` entre celles-ci.

```
mkdir /data ; touch /data/file1 ; cat /home/bonjour.txt
mkdir: cannot create directory `./data': Permission denied
touch: cannot touch `./data/file1': No such file or directory
Bonjour le monde !
```

Dans cet exemple, l'utilisateur n'a pas le droit de créer un répertoire à la racine. La commande de création d'un ficbhier dans le répertoire échoue aussi. La dernière commande (sans lien avec les précédentes) aboutit. Dans un script, le traitement des commandes est séquentiel. Le `;` est substitué par un retour à la ligne.

```
mkdir /data ; touch /data/file1 ; cat /home/bonjour.txt
```

Donne:

```
mkdir /data
touch /data/file1
cat /home/bonjour.txt
```

Conditionnel

L'opérateur `&&` conditionne l'enchaînement de commandes au résultat d'exécution positif de la commande précédente.

- Cette structure peut se traduire par : Si la **commande aboutit alors** je fais la suivante.

```
mkdir /data && touch /data/file1
mkdir: cannot create directory `./data': Permission denied
```

```
mkdir ./data && touch ./data/file1
ls -l ./data/
-rw-r--r-- 1 Toto Toto 0 2021-12-26 02:12 file1
```

Le système s'appuie sur le code retour de la commande précédente (variable réservée `$?`) : Si `$?` = `0` (pas d'erreur) : alors la commande suivante est exécutée Si `$?` ≠ `0` (erreur) : alors la commande suivante n'est pas exécutée

L'opérateur `||` conditionne l'enchaînement de commandes à l'échec d'exécution de la commande précédente.

- Cette structure peut se traduire par : Si la **commande n'a pas abouti alors** je fais la suivante.

```
mkdir ./data 2>/dev/null || echo "Erreur de création du
répertoire 'data'"
Erreur de création du répertoire 'data'
```

Regroupements de commandes

Les regroupements de commandes peuvent être utilisés pour: Executer plusieurs commandes dans un même environnement. Rediriger le résultat de plusieurs commandes vers un fichier ou un tube.

Regroupement avec création de sous shell

Deux syntaxes de regroupement de commandes sont utilisables: `(`regroupement parenthèses`)` et `{`regroupement accolade`}`

Pour regrouper dans un environnement enfant l'exécution de plusieurs commandes, on utilise la syntaxe suivante :

```
(cmd1 ; cmd2)
```

Pour plus de lisibilité, nous écrirons ainsi:

```
(
    cmd1
    cmd2
)
```

Regroupement sans création de sous shell

Pour regrouper dans un environnement enfant l'exécution de plusieurs commandes, on utilise la syntaxe suivante :

```
{cmd1 ; cmd2}
```

Pour plus de lisibilité, nous écrirons ainsi:

```
{
    cmd1
    cmd2
}
```

Les couleurs dans le shell

Il est possible d'utiliser quelques couleurs dans le shell ou dans les scripts. Elles peuvent être utiles pour mettre en valeur des informations importantes.

Couleur	Caractères	Fond	Attribut des caractères
Noir	30	40	0 : Aucun
Rouge	31	41	1 : Gras
Vert	32	42	2 : Souligné
Jaune	33	43	7 : Inversé
Bleu	34	44	8 : Invisible
Magenta	35	45	9 : Barré
Cyan	36	46	
Gris Clair	37	47	

Pour coloriser une information, les codes couleur seront à encadrer pour : Annoncer la couleur : `\033[` ou `\e[` Finir la définition de couleur : `m` Remettre la valeur initiale : `\033[0m`

```
echo -e "voici du \033[1;32mvert\033[0m"
voici du vert
```

Execution Conditionné

Dans de nombreux cas pratiques, la réalisations d'actions est conditionnée.

- Exemple: La valeur de la variable `nb` est conditionnée
    - Au nombre d'arguments passé au script
    - Puis à la présence ou non de contenu dans cette variable.

En Scripting Shell, pour traduire cet algorithme, nous utiliserons: Une structure de contrôle comme la structure `if` Une expression de test (qui sera intégrée) à la structure de contrôle. Ainsi, la première condition serait traduite ainsi: Structure de contrôle de `if`:

```
if condition ;then
    nb=$1
fi
```

Test de la condition:

```
[[ $nbarg -eq 1 ]]
```

Structure complète:

```
if [[ $nbarg -eq 1 ]] ; then
nb=$1
fi
```

L'instruction `if`est utilisée pour structurer la réalisation conditionnée de commandes.

```
si
    la condition est remplie
    alors je fais ceci
    sinon je fais cela
fin du si
```

Cette instruction requiert au minimum :

- Un début : `if`
- L'exécution d'une commande (un test de condition par exemple)
- L'exécution d'au moins une action si la condition est vérifiée : `then`
- Une fin : `fi` Cette instruction peut être complétée par:
- L'exécution d'autres actions si la condition est vérifiée
- L'exécution d'actions à réaliser si la condition n'est pas vérifiée: `else`

Exemples:

```
if [[ "$LOGNAME" = root ]] ; then
echo "Ne pas se connecter en root"
fi
```

```
if ls /tmp ; then
echo "/tmp existe"
fi
```

```
if [[ "$LOGNAME" = root ]] ; then
echo "Ne pas se connecter en root"
```

```
else
echo "Bienvenue sur la machine $HOSTNAME"
fi
```

```
if [[ "$LOGNAME" = root ]] ; then
echo "Ne pas se connecter en root"
echo "Le script va quitter"
exit 1
else
echo "Bienvenue sur la machine $HOSTNAME"
echo "Le script continue"
fi
```

L'imbrication du `if` :

```
if <condition> ; then
    <action>
else
    if <condition> ; then
        <action>
    else
<       action>
    fi # fin du second if
fi # fin du premier if
```

L'utilisation du `elif`:

```
if <condition> ; then
    <action>
elif <condition> ; then
    <action>
else
    <action>
fi # fin commune des 2 if
```

Les Boucles

- Une boucle est une structure permettant de répéter plusieurs fois un même bloc d'actions.
- Le nombre de fois à exécuter le bloc d'actions peut être déterminé par une condition ou une liste d'objets à traiter.
- Quand il s'agit d'une condition, penser à faire évoluer la valeur des variables utilisées dans celle-ci (notamment au moyen de calcul)

`expr`: Calcul arithmétique

- Les arguments passés à la commande peuvent être des entiers ou des variables (contenant des entiers).
- Chaque opérateur doit être précédé et suivi d'un espace.

```
expr 2 + 7
9
expr 2 \* 7
14
```

Les commandes `let` et `((…))` peuvent être utilisées en remplacement de la commande `expr`.

La boucle *while*

Structure de bouclage basée sur une condition atteinte : tant que la condition est atteinte, on effectue le bloc d'actions.

Exemple:

```
while [[ -z "$nom" ]] ; do
echo -e "Veuillez entrer votre nom : \c"
read nom
done
echo "Bonjour $nom"
```

- La structure while analyse le code retour de la condition pour déterminer si le bloc d'actions est à exécuter ou s'il doit sortir de la boucle.
- Généralement, la condition se matérialise par une commande de test.
- On agit alors dans le bloc d'actions afin de faire évoluer le résultat du test.
- Cependant, toute commande peut être utilisée en condition (tant que son code retour est « 0 », on reste dans la boucle).

Cas particulier des boucles infinies

- Pour générer une boucle infinie, on utilise la commande `true` ou `:`.
- Les boucles infinies sont notamment utilisées pour la création de menus.

Exemple:

```
while true ; do
    echo menu # Affichage du menu
    echo "1) copie des fichiers"
    echo "2) restauration des fichiers"
    echo "q) Quitter"
    # Récupérer la saisie de l'utilisateur dans $choix
    read -p "Taper 1, 2 ou q pour continuer : " choix
    # Tester la valeur de $choix
    case $choix in
        1) echo copie des fichiers; […] ;;
```

```
        2) echo restauration des fichiers; […] ;;
        q) clear ; exit 0 ;;
        *) echo -e "\e[41msaisie incorrecte \e[0m" ;;
    esac
done
```

La boucle *until*

Structure basée sur une condition à atteindre : tant que la condition n'est pas atteinte, on effectue un bloc d'actions. La vérification de la condition mène à la sortie de la boucle. La structure `until` et son mode de fonctionnement sont identiques à la boucle `while`. Seule la condition de bouclage les différencie. Pour faire une boucle infinie avec `until`, il est possible d'utiliser la commande `false`.

```
until [[ -n "$age" ]] ; do
    echo -e "Saisissez votre age : \c"
    read age
done
```

La boucle *for*

La boucle `for` peut boucle: Pour un ensemble de valeurs à traiter Un nombre de fois prédéterminé

- La syntaxe d'utilisation suivante permet de boucler pour un ensemble de valeurs à traiter.

```
for element in LISTE DE VALEURS ; do
    ACTIONS
done
```

- Lors du premier passage dans la boucle, la valeur affectée à la variable définie `$element` sera la première valeur de la liste : « `LISTE` ».
- Lors du second passage, `$variable` contiendra « `DE` » et ainsi de suite jusqu'à ce que les valeurs aient été traitées.

Boucler pour un ensemble de valeurs

```
$ for valeurs in "petit exemple" de "boucle for" ; do
>echo "Entrée dans le bloc d'actions de la boucle"
>echo "contenu de la variable valeurs : $valeurs"
>done
Entrée dans le bloc d'actions de la boucle
contenu de la variable valeurs : petit exemple
Entrée dans le bloc d'actions de la boucle
contenu de la variable valeurs : de
```

```
Entrée dans le bloc d'actions de la boucle
contenu de la variable valeurs : boucle for
```

La liste de valeurs est intéprétée par le shell. La liste des caractères de séparation de champs entre les différentes valeurs est récupérée depuis la variable d'environnement `IFS` (Internal Field Separator). Il est parfois utile de modifier cette valeur.

```
$ set | grep ^IFS
IFS=$' \t\n'
```

Exemples

```
cat fic | while read nom ; do
    if [[ "$nom" = toto ]] ; then
        echo "$nom"
            read -p "message : " choix </dev/tty
            echo "$choix"
    fi
done
```

Liste et Calcule le nombre et pourcentage de dossier/fichier présents dans un dossier selon un fichier d'extensions

```
if [[ $# -ne 2 ]]; then
      echo -e " Syntaxe: $0 <Chemin du dossier à traiter> <Fichiers extensions>"
      exit 1
fi

dossier=$1 #Variable Chemin à traiter
fichier=$2 #Variable fichier extensions

if [[ -f $2 ]]; then    # Vérification de la présence des 2 arguments
      nbfiles=$(ls ${dossier} | wc -l ) # Liste et compte le nombre de fichier
dans le dossier
      echo -e "Nombre total de fichier dans $(basename "$dossier") : ${nbfiles}"

    nbdirs=$(find "${dossier}" -type d | wc -l) # Liste et Compte le nombre de
dossier et calcule le pourcentage
    pourcentage_dirs=$(echo "scale=2; ${nbdirs} * 100 / ${nbfiles}" | bc)
    echo ""
    echo -e "Nombre total de dossiers dans le dossier $(basename "$dossier") :
${nbdirs}"
    echo ""
    echo -e "${pourcentage_dirs}% des fichiers du répertoire $(basename
"$dossier") sont des dossiers\n"
```

```
      for ext in $(cat ${fichier}) # Boucle for pour chaque fichier, trouver les
occurences des extensions
        do
            nbficext=$(ls ${dossier}/*.${ext} 2>/dev/null | wc -w) # Liste et
compte le nombre de fichier
            if [[ ${nbficext} -gt 0 ]]; then # Si supérieur à 0 alors
                echo "----------------------"
                echo -e "${ext}"
                echo "----------------------"
                echo "-> ${nbficext} fichiers ${ext}"
                pourcentage=$(expr ${nbficext} \* 100 \/ ${nbfiles})
                echo "-> $pourcentage% des fichiers du répertoire $(basename
"$dossier")"
            else
                echo "----------------------"
                echo -e "${ext}"
                echo "----------------------"
                echo -e " Aucun fichier ${ext} dans $(basename "$dossier")"
            fi
        done
else
    echo " Le fichier ${fichier} contenant les extensions n'a pas été trouvé. "
    exit 2
fi
```

Calculateur d'année bissextile selon une plage d'année

```
echo -e " ----\e[1mCalculateur de nombre d'années bissextiles\e[0m---- "
echo ""
read -p "Entrez l'année de début : " debut
read -p "Entrez l'année de fin : " fin

count=0

for ((i=debut;i<=fin;i++))
do
    if [ $((i%4)) -eq 0 ] && [ $((i%100)) -ne 0 ] || [ $((i%400)) -eq 0 ]
    then
        ((count++))
    fi
done

echo "Il y a $count années bissextiles entre $debut et $fin."
```

Script intéractif de création, modification, suppression et vérification d'un compte utilisateur

```
if [[ $EUID -ne 0 ]]; then
   echo "Ce script doit être exécuté en tant que root."
   exit 1
```

```
fi

while true; do
echo -n " Saisir le compte utilisateur souhaité: "
read user

if [[ "${user,,}" = "q" ]]; then
        echo "Au revoir !"
        exit 0
fi

echo ""
echo " GESTION DES UTILISATEURS: $user "
echo " _____ "
echo ""
echo " C - Créer le compte utilisateur "
echo " M - Modifier le mot de passe utilisateur "
echo " S - Supprimer le compte utilisateur "
echo " V - Vérifier si le compte utilisateur existe "
echo ""
echo " Q - Quitter "
echo ""
echo -n " Veuillez saisir votre choix: "
read choix

echo ""
case $choix in
    c|C)
    if useradd -m -s /usr/bin/zsh $user ; then
        echo -e " Le compte $user a été créé "
    fi
    ;;
    m|M)
    if passwd $user ; then
    echo -e "Modification du mot de passe de $user avec succès."
    fi
    ;;
    s|S)
    if deluser --remove-home $user ; then
        echo -e " Suppression du compte avec succès. "
    fi
    ;;
    v|V)
        echo "Verification de l'existence du compte"
        if grep -qi "$user" /etc/passwd; then
            echo -e " Compte utilisateur $user \033[1;32mEXISTANT \033[0m "
            echo ""
        else
            echo -e " Compte utilisateur $user \033[1;31mINEXISTANT \033[0m "
            echo ""
        fi
    ;;
    q|Q)
        exit 0
```

```
    ;;
    *)
        echo " Choix non pris en charge. Saisissez: C ; M ; S ; V ou Q "
    ;;
esac

done
```

Récupère et affiche sous forme de tableau les users présents dans /etc/passwd

```
while IFS=: read -r user uid gid desc home shell; do
    uid=$(echo "$user" | awk '{print $1}')
    home=$(echo "$user" | grep '/home')

    if [[ -n "$home" ]]
    then
        desc=$(echo "$user" | awk -F: '{print $5}')
        shell=$(echo "$user" | awk '{print $7}')

        echo "---------------------------------"
        echo " Identifiant : $uid "
        echo "- - - - - - - - - - - - - - - - - - - - -"
        echo "UID : $(id -u "$uid") GID : $(id -g "$uid")"
        if [[ -n "$desc" ]]
        then
            echo "Description : $desc"
        fi
        echo "Répertoire personnel : $home"
        echo "Shell : $shell"
    fi
done < <(cat /etc/passwd | sed -e 's/::/:x:/g' -e 's/:/\ /g')
```

Les Fonctions

Les fonctions permettent de regrouper plusieurs commandes qui pourront être exécutées de façon régulière durant le script.

Il est conseillé d'utiliser des minuscules dans le nom des fonctions et de préfixer le nom de la fonction avec le terme « `function_` » ou « `func_` » suivi du nom réel de la fonction.

Cela permet de lire le script plus facilement et savoir quand on fait appel à une fonction ou à une commande.

- La mise en œuvre de fonction se fait en deux étapes :
- Déclaration de la fonction et de son contenu
- Appel de la fonction
- Les déclarations de fonctions peuvent être stockées dans un fichier séparé, pour rendre un script plus lisible et les réutiliser dans d'autres scripts.
- Les fonctions peuvent être invoquées avec des paramètres, comme les commandes et les scripts.

Syntaxe de déclaration d'une fonction

```
nomfonction()
{
ACTIONS
}
```

Déclaration et utilisation dans le script

```
#!/bin/bash
func_infos() {
    echo "Informations sur la distribution utilisée"
    lsb_release -dric
}
# Afficher les infos sur l'OS
func_infos
```

Les variables dans les fonctions

```
#!/bin/bash
nbr=50
func_majnbr () {
    nbr=100
}
echo "variable nbr avant fonction: $nbr"
func_majnbr
echo "variable nbr après fonction : $nbr"
```

Passage de paramètres aux fonctions

```
#!/bin/bash
func_accueil () {
    echo "Message d'accueil :"
    echo "Bonjour $1 $2"
    echo "Bienvenue sur la machine $HOSTNAME"
}
read -p "Entrez votre prénom nom (ex Marc Dubois) : " prenom nom
func_accueil "$prenom" "$nom"
```

Déclaration dans un fichier indépendant

- Les fonctions peuvent être définies dans un fichier indnt du script.
- Il suffit ensuite d'appeler le fichier avec les commandes `source` ou « . » (point), afin que le contenu de ce fichier soit interprété dans l'environnement d'exécution courant.

- Avantages de sortir les fonctions des scripts :
- Le script est plus lisible (... ou pas, en fonction des cas)
- Le script est plus facile à déboguer
- Les fonctions peuvent être utilisées par d'autres scripts

Contenu du fichier de fonctions « mesfonctions »:

```
func_accueil () {
echo "Message d'accueil :"
echo "Bonjour $1 $2"
echo "Bienvenue sur la machine $HOSTNAME"
}
```

Le fichier de fonction n'a pas besoin d'être exécutable, des privilèges de lecture suffisent.

Contenu du script:

```
#!/bin/bash
# Déclaration des fonctions
source ~/mesfonctions
read -p "Saisissez votre prénom et votre nom (ex Marc Dubois) : " prenom
nom
# on appelle la fonction accueil du fichier ~/mesfonctions
func_accueil "$prenom" "$nom"
```