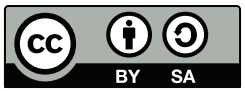
Api/casoft Init - Jour 3 (matin) - Scripts shell

Rémy Huet Quentin Duchemin

Association Picasoft

Mercredi 29 janvier 2020



Qu'est ce qu'un shell ?

- Un shell est un **interpréteur de commandes**.
- C'est le programme qui gère l'**invité de commandes**.
- C'est donc lui qui a exécuté toutes les commandes que vous avez pu taper ces derniers jours.

Les différents shells

Présentation

sh

sh ou *Bourne Shell* est un des shells les plus anciens (1977). Il est sur tous les OS basés sur UNIX, mais est le plus pauvre en terme de fonctionnalités. Il a notamment introduit les *pipes*.

Bash

Bash ou *Bourne Again Shell* est une amélioration de sh. Il est le shell par défaut de la plupart des distributions Linux et des terminaux Mac OS (et celui que vous utilisez !).

Mais aussi

Il existe d'autres shells dérivant de sh : csh, tcsh, ksh, zsh, ainsi que d'autres shell complètement différents (xonsh...).

Les différents shells

Connaître son shell

« Et moi, j'ai quoi comme shell ? »

On peut connaître son shell par défaut grâce à la variable d'environnement `$SHELL`. On l'affiche donc avec `echo $SHELL`.

On peut aussi voir la liste des shells installés : ils sont généralement dans le répertoire `/bin`.

```
remy@hp-remy:~$ echo $SHELL
/bin/bash
remy@hp-remy:~$ ls /bin/*sh
/bin/bash  /bin/csh  /bin/sh  /bin/tcsh  /bin/zsh
```

Shell et script

Ou l'art de ne plus taper les commandes 10 fois d'affilé

Qu'est ce qu'un script ?

Un script est tout simplement un **programme écrit dans le langage du shell**. Il peut contenir une suite de commandes, mais aussi des « structures de contrôle » (conditions, boucles...).

On stocke les scripts dans des fichiers, ce qui permet :

- D'automatiser des tâches, sans avoir à ré-écrire les commandes à chaque fois
- D'écrire des instructions sur plusieurs lignes de manière lisible
- D'éviter de se perdre dans l'historique pour retrouver comment on a fait telle ou telle chose

Un tout premier script !

Hello World !

Pour nos scripts, on utilisera Bash, qui est un bon compromis entre puissance et compatibilité.
On commence par créer un fichier :

```
remy@hp-remy:~$ mkdir scripts && cd scripts
remy@hp-remy:~/scripts$ nano helloWorld.sh
```

Par convention, on utilisera l'extension **.sh** pour tous nos scripts.
On écrit ensuite dans le fichier :

```
#!/bin/bash
echo 'Hello World !'
```

On rend enfin le fichier executable et on lance le script :

```
remy@hp-remy:~/scripts$ chmod +x helloWorld.sh
remy@hp-remy:~/scripts$ ./helloWorld.sh
Hello World !
```

Le sha-bang

« Attends, mais c'est quoi cette ligne ? »

```
#!/bin/bash <----------
```

Le sha-bang

Le #! est appelé **sha-bang** (ou shebang). Il est facultatif, mais permet de s'assurer que c'est bien le bon shell qui exécutera notre programme
On aurait pu évidemment mettre à la place `/bin/sh` ou `/bin/tcsh`.

Qu'est-ce qu'une variable ?

Définition

Une variable est une zone de mémoire dans laquelle on stocke une valeur. Concrètement, on ne s'occupera pas de la partie gestion de mémoire, on se limitera au couple clé (nom de la variable)/valeur.

Pourquoi variable ?

On appelle cela une variable, car la valeur associée à la clé peut changer au cours de l'execution du programme. On dit qu'on peut la **réaffecter**.

Affecter une variable

Affecter

Affecter une valeur à une variable, c'est dire « *Quand j'utiliserai le nom de cette variable, tu le remplaceras par cette valeur* ».

Syntaxe

Pour affecter une valeur à une variable, on utilisera la syntaxe :

nom=valeur

Attention !

Il ne faut surtout pas mettre d'espace avant et après le signe =

Exemple :

Dans le fichier `script.sh` on écrit :

```
#!/bin/bash
ma_variable="Salut tout le monde"
```

Affecter une variable

On exécute le script :

```
remy@hp-remy:~/scripts$ chmod +x script.sh
remy@hp-remy:~/scripts$ ./script.sh
```

« Attends, mais pourquoi y a rien qui s'affiche ? »

Réponse

Parce que notre script ne fait qu'**affecter** une valeur à une variable ! On ne lui a jamais demandé d'afficher quoi que ce soit !

Lire une variable

Accéder au contenu

Pour accéder au contenu d'une variable, il faut écrire son **nom**, précédé du symbole **$**.

On modifie le fichier `script.sh` :

```
#!/bin/bash
ma_variable="Salut tout le monde"
echo $ma_variable
```

En exécutant le script, on obtient :

```
remy@hp-remy:~/scripts$ ./script.sh
Salut tout le monde
```

Les quotes

Explications

Il y a trois types de « quotes » en Bash :

- Les « simple quotes » (') qui affichent le contenu tel qu'il est écrit
- Les « double quotes » (") qui interprètent des variables ou des caractères d'échapement dans du texte
- Les « backquotes » (`) qui interprètent le contenu comme une commande

On écrit dans `script.sh`

```
#!/bin/bash
cmd="ls"

echo '$cmd'
echo "$cmd"
echo `$cmd`
```

Le résultat obtenu est :

```
$cmd
ls
helloWorld.sh script.sh
```

Explications

Ligne 1 L'utilisation de simple quotes n'interprète pas `$cmd` comme une variable.

Ligne 2 L'utilisation de double quotes interprète `$cmd` comme une variable et affiche son contenu.

Ligne 3 L'utilisation de backquotes interprète `$cmd` comme une commande et execute donc le `ls` depuis le répertoire de lancement du script.

Quelle forme utiliser pour les variables ?

On préférera utiliser des double quotes (") quand on veut utiliser le contenu des variables, surtout si le contenu peut contenir des espaces !

Entrées utilisateur

Contexte

L'intérêt des variables, c'est aussi de pouvoir demander à l'utilisateur de saisir des données pour rendre le programme **interactif**.

Utilisation

Pour lire une entrée utilisateur, on utilise la commande

`read variable`

L'option `-p` permet d'afficher à destination de l'utilisateur.

RTFM pour plus de super options !

Exemple (toujours dans `script.sh`) :

```
read -p 'Quel est votre nom ? ' nom
echo "Salut $nom"
```

Les opérations arithmétiques

Contexte

Bash ne gère pas nativement les variables comme des nombres (par exemple, « 2+2 » renverra une erreur). Il faut passer par la commande `let`.

Exemple :

```
#!/bin/bash
let "a=5"
let "b=2"
let "c=a+b"
echo $c
```

Quand on execute le script, on obtient :

```
remy@hp-remy:~/scripts$ ./script.sh
7
```

Les opérateurs arithmétiques

Les opérateurs reconnus par `let` sont :

Symbole	Signification	Exemple
+	Addition	5+2=7
-	Soustraction	5-2=3
*	Multiplication	5*2=10
/	Division euclidienne	5/2=2
**	Puissance	5**2=25
%	Modulo	5%2=1

Table – Les opérateurs supportés par let

Info

Comme pour le langage C, il est possible d'utiliser la syntaxe `let "a+=3"` à la place de `let "a=a+3"` (ou autres opérateurs).

Les paramètres d'exécution

Contexte

On a vu cette semaine qu'une commande peut recevoir un ou plusieurs paramètres, séparés par des espaces (exemple : `mv <source> <destination>`). Il en est de même pour les scripts Shell.
Comme pour les commandes, l'ordre des paramètres est **primordial** !

Accéder aux paramètres

Lors de l'exécution, des variables sont automatiquement créées : $# contient le nombre de paramètres, $1 le premier, $2 le deuxième . . . et $9 le neuvième.

Exemple :

```
#!/bin/bash
echo "Il y a $# paramètres d'execution"
echo "Le premier paramètre est $1"
```

Ce qui donnera :

```
remy@hp-remy:~/scripts$ ./script.sh test
Il y a 1 paramètres d'execution
Le premier paramètre est test
remy@hp-remy:~/scripts$ ./script.sh test toto
Il y a 2 paramètres d'execution
Le premier paramètre est test
remy@hp-remy:~/scripts$ ./script.sh
Il y a 0 paramètres d'execution
Le premier paramètre est
```

« Mais attends, je peux avoir que 9 paramètres ? »

Non, on peut en fait avoir plus de paramètres, bien qu'en général on utilise plutôt des fichiers de configuration quand le nombre de paramètres explose. On en reparlera quand on saura utiliser les boucles, pour faire ça *proprement*.

Les tableaux

Créer et lire un tableau

Contexte

En Bash, on peut également déclarer des tableaux. Ce sont des variables un peu spéciales, qui contiennent plusieurs valeurs référencées par un **index**.

Accéder à une valeur

Les valeurs stockées dans un tableau possèdent un **index** qui permet d'y accéder individuellement. La syntaxe pour lire une valeur est `${nom[index]}`. On peut également utiliser `${nom[@]}` pour afficher toutes les valeurs associées à un index **numérique**.

Différents types de tableaux

- Les tableaux **numériques**, où les indices sont des nombres,
- Les tableaux **associatifs**, où les indices sont des mots arbitraires.

Les tableaux

Tableau numérique

Créer un tableau numérique

On utilise la syntaxe `tab=(el0 el1 el2)`. Dans cet exemple, le tableau contient trois éléments, accessibles aux indices **0**, 1 et 2. On peut par exemple accéder au premier élément avec `$tab[0]`.

Modifier une valeur

La syntaxe `tab[index]` permet également de modifier une valeur, par exemple en écrivant `nom[index]=valeur`.

Ajouter ou supprimer une valeur

On peut ajouter une valeur en utilisant la syntaxe précédente, par exemple : `tab[3]=el3`. Le plus pratique reste d'utiliser la syntaxe `tab+=(el3 el4 el5)`, qui prend automatiquement les bons indices. Pour supprimer une valeur, on utilisera la syntaxe `unset tab[index]`.

Les tableaux

Tableau associatif

Créer un tableau associatif

On utilise la syntaxe `declare -A tab`.

Opérations classiques

On utilise la même syntaxe que pour les tableaux numériques, excepté pour ajouter plusieurs éléments d'un seul coup avec le +=.
Et cette fois ci, les indices peuvent être des mots, ce qui est très pratique pour stocker des scores, des clé-valeurs...

Remarque

Il est théoriquement possible de mélanger tableau numérique et tableau associatif, mais franchement, c'est une mauvaise idée...

Les tableaux

Exemple

On ecrit dans `script.sh` :

```
#!/bin/bash

tab_num=( e0 e1 )
echo $tab_num[0]
echo $tab_num[@]
unset tab_num[0]
echo $tab_num[*]

declare -A tab_ass
tab_ass[foo]=jury
tab_ass[bar]=bc01
echo $tab_ass[foo]
echo $tab_ass[@]
```

On exécute, et on obtient :

```
e0
e0 e1
e1
jury
jury bc01
```

Introduction

Définition

On appelle **condition** ou **structure conditionnelle** un morceau de code qui est exécuté uniquement dans certains cas.

En algorithmique, on les voit souvent sous la forme :

```
SI condition
ALORS
    actions
FIN_SI
```

Et en Bash ?

En Bash, la structure à utiliser est la suivante :

```
if [ condition ]
then
    actions
fi
```

Attention !

Les espaces entre les `[]` et la condition sont **indispensables**.

- À noter : `fi`, c'est `if` à l'envers !
- On verra plus tard quelles sont les conditions que l'on peut utiliser.

Si . . . sinon . . .

Contexte

La structure précédente permet de réaliser des actions si une condition est évaluée à **vrai**. On rencontre souvent une autre structure, qui dit aussi quoi faire si la condition est évaluée à **faux** :

```
SI condition
ALORS
    actions
SINON
    actions
FIN_SI
```

En Bash ...

En Bash, on utilise la même syntaxe, et on utilise « `else` » comme mot clé pour le « sinon ».

```
if [ condition ]
then
    actions
else
    actions
fi
```

Si . . . sinon si . . . sinon

« Et si je veux faire encore plus compliqué ? »

Il existe encore un autre moyen d'écrire des `SI` :

```
SI condition
ALORS
    actions
SINON SI condition
ALORS
    actions
SINON
    actions
FIN_SI
```

On peut mettre autant de « SINON SI . . . ALORS . . . » que l'on veut !

En Bash

En bash, « SINON SI » s'écrit `elif` (contraction de `else` et `if`) :

```
if [ condition ]
then
    actions
elif [ condition ]
then
    actions
else
    actions
fi
```

Chaînes de caractères

Les différents tests

Test et syntaxe	Explications
`"$chaine1" == "$chaine2"`	Teste si `$chaine1` et `$chaine2` sont **égaux**
`"$chaine1" != "$chaine2"`	Teste si `$chaine1` et `$chaine2` sont **différents**
`-z $chaine`	Teste si la la variable `$chaine` est **vide**
`-n $chaine`	Teste si la variable `$chaine` est **non vide**

Table – Les tests sur les chaînes de caractères

Chaînes de caractères

Exemples

Dans `script.sh`, on écrit :

```
#!/bin/bash

if [ "$1" == "mot_de_passe" ]
then
    echo 'Bienvenue'
elif [ "$1" == "password" ]
then
    echo 'Welcome !'
else
    echo 'Raté !'
fi

if [ -z $2 ]
then
    echo "Il n'y a pas de second argument"
fi
```

L'exécution du script donne :

```
remy@hp-remy:~/scripts$ ./script.sh
Raté !
Il n'y a pas de second argument
remy@hp-remy:~/scripts$ ./script.sh mot_de_passe
Bienvenue
Il n'y a pas de second argument
remy@hp-remy:~/scripts$ ./script.sh password toto
Welcome !
```

Nombres

Les différents tests

Test et syntaxe	Explications
`$a -eq $b`	Test d'égalité
`$a -ne $b`	Test d'inégalité
`$a -lt $b`	$a inférieur à $b
`$a -le $b`	$a inférieur ou égal à $b
`$a -gt $b`	$a supérieur à $b
`$a -ge $b`	$a supérieur ou égal à $b

Table – Comparaisons sur les nombres en Bash

Nombres

Exemple

Dans le fichier `script.sh`, on écrit :

```
#!/bin/bash

read -p 'Quel est ton age ? ' age

if [ $age -lt 18 ]
then
    echo 'Tu es mineur'
fi

if [ $age -ge 18 ]
then
    echo 'Tu es majeur'
    if [ $age -eq 18 ]
    then
        echo 'De peu !'
    fi
fi
```

On exécute le script :

```
remy@hp-remy:~/scripts$ ./script.sh
Quel est ton age ?   14
Tu es mineur

remy@hp-remy:~/scripts$ ./script.sh
Quel est ton age ?   21
Tu es majeur

remy@hp-remy:~/scripts$ ./script.sh
Quel est ton age ?   18
Tu es majeur
De peu !
```

On note bien que `-lt` a vérifié que l'âge était **strictement** inférieur tandis que -ge a vérifié qu'il était supérieur ou égal.

Fichiers

Les différents tests

Test et syntaxe	Explications
`-e $nom`	Le fichier (ou répertoire) `$nom` existe
`-d $nom`	`$nom` est un répertoire
`-f $nom`	`$nom` est un fichier
`-L $nom`	`$nom` est un lien
`-r $nom`	`$nom` est lisible
`-w $nom`	On peut écrire dans `$nom`
`-x $nom`	On peut exécuter `$nom`
`$f1 -nt $f2`	`$f1` est plus récent que `$f2`
`$f1 -ot $f2`	`$f1` est plus vieux que `$f2`

Table – Tests sur les fichiers en Bash

Fichiers

Exemple

```
#!/bin/bash

mkdir -p dossier
if [ -d dossier ]
then
    echo 'dossier est un répertoire'
    touch dossier/test.sh
    if [ -w dossier/test.sh ]
    then
        if [ -x dossier/test.sh ]
        then
            echo Fichier executable
        else
            echo 'rendre le fichier executable ...'
            chmod a+x dossier/test.sh
            echo 'Écrire dans test.sh'
            echo '#/bin/bash' > dossier/test.sh
            echo 'ls /' >> dossier/test.sh
        fi
    fi
    echo 'Exécuter dossier/test.sh'
    ./dossier/test.sh
else
    echo 'Erreur'
    exit 1
fi
```

Résultat :

```
remy@hp-remy:~/scripts$ ./script.sh
dossier est un répertoire
Fichier executable
Écrire dans test.sh
Exécuter dossier/test.sh
bin  boot  cdrom  core dev  etc  home initrd.img
initrd.img.old  lib  lib64 lost+found  media  mnt
opt  proc  root  run  sbin  srv  sys  tmp usr var  vmlinuz
```

Regardez bien ces lignes-là :

```
if [ -w dossier/test.sh ]
    then
        if [ -x dossier/test.sh ]
        then
            echo Fichier executable
        else
            echo 'rendre le fichier executable ...'
            chmod a+x dossier/test.sh
        [...]
        fi
```

Est-on obligé d'imbriquer les `if` ? Ne pourrait-on pas inverser les tests pour éviter les `else` quand c'est justement le seul cas que l'on veut gérer ?

Les opérateurs logiques

Contexte

Les tests que nous utilisons depuis tout à l'heure renvoient des valeurs logiques (vrai ou faux). Sur ces valeurs, on peut utiliser des **opérateurs logiques**.

Opérateur	Signification	Vrai si ...
&&	ET	Les deux conditions sont vérifiées
||	OU	Une des conditions est vérifiée
!	NON	La condition est fausse

Table – Les opérateurs logiques

Pour utiliser ces opérateurs, on écrira [! condition] pour NON ou [condition1] <opérateur> [condition2] pour les autres.

Exemple

Dans `script.sh` :

```
#!/bin/bash

age1=17
age2=19

if [ $age1 -ge 18 ] && [ $age2 -ge 18 ]
then
    echo "Les deux sont majeurs"
fi

if [ $age1 -lt 18 ] || [ $age2 -lt 18 ]
then
    echo "Un des deux n'est pas majeur"
fi

if [ ! $age1 -lt 17 ]
then
    echo "age1 >= 17"
fi
```

On exécute :

```
remy@hp-remy:~/scripts$ ./script.sh
Un des deux n'est pas majeur
age1 >= 17
```

Introduction

Contexte

Lorsqu'on veut réaliser des actions différentes en fonction de la valeur d'une seule variable, on peut faire un grand `if` avec beaucoup de `elif`.
Néanmoins, cela est assez lourd à écrire et peu lisible.
Dans ce genre de cas, on utilisera une autre structure, le `case`, qui permet d'alléger grandement le code.

```
#!/bin/bash
case $extension_ficher in
    '.sh')
        echo ``C'est un script ''
        ;;
    '.tex')
        echo ``C'est un fichier LaTeX ''
        ;;
    '.doc' | '.xls' | '.ppt')
        echo 'Sans déconner ?'
        ;;
    *)
        echo 'Je ne connais pas !'
        ;;
esac
```

Explications

```
case $exension_fichier in
```

On teste la valeur de la variable `$extension_ficher` (que l'on suppose exister).

```
'.sh')
    echo ``C'est un script ''
    ;;
```

Si c'est `.sh`, on affiche « C'est un script ! ». Les deux `;;` signifient que c'est la fin de ce qu'il faut exécuter, et on sort du case.

```
'.doc' | '.xls' | '.ppt')
        echo 'Sans déconner ?'
        ;;
```

On peut effectuer la même action pour plusieurs valeurs.
Attention le « ou » s'écrit alors avec un seul `|`.

Explications

```
*)
        echo 'Je connais pas !'
        ;;
```

« * » est un « wildcard », ou joker. Il capture toutes les possibilités. Ici, tout ce qui n'est pas passé par une autre condition rentre dans ce cas. C'est l'équivalent du « `else` ».

```
esac
```

C'est « `case` » à l'envers !

Introduction

On sait désormais effectuer des actions diverses, et ce en fonction de paramètres donnés au programme par l'utilisateur.
Maintenant, comment faire si l'on veut effectuer la même opération un certain nombre de fois ?

Définition

Une boucle est une structure permettant de répéter un ensemble d'instructions. Elle est composée :

- D'un corps : c'est l'ensemble des instructions qui seront exécutées lors d'un passage dans la boucle.
- D'une condition de réalisation : si celle-ci est vraie, une **itération** est effectuée. Si la condition est toujours vraie, on crée une boucle. . . infinie.

En théorie

Explications

Elle est composée d'une condition et d'un corps. Elle est de la forme :

```
TANT QUE condition
FAIRE
    actions
FIN_TANT_QUE
```

Concrètement

Son fonctionnement est assez simple. Tant que la condition est vérifiée, la boucle s'exécute.

Et en Bash ?

En bash, la syntaxe de cette boucle est la suivante :

```
while [ condition ]
do
    actions
done
```

Exemple. Dans `script.sh` :

```
#!/bin/bash

while [ "$reponse" != 'oui' ]
do
    read -p "Dites oui : " reponse
done
```

Ce script demandera à l'utilisateur d'entrer du texte tant que l'utilisateur n'enverra pas « oui ».
On notera que comme pour les if, on peut utiliser les opérateurs logiques.

En théorie

Utilité

La boucle pour permet de parcourir une liste de valeurs et de boucler autant de fois qu'il y a de valeurs.

Syntaxe :

```
POUR variable PRENANT valeur1 valeur2 valeur3
FAIRE
    actions
FIN_POUR
```

Et en bash ?

En bash, la syntaxe est la suivante :

```
for variable in val1 val2 val3
do
    actions
done
```

Exemple :

```
#!/bin/bash

for file in *
do
    cat $file
done
```

Ce script affichera l'ensemble des fichiers du répertoire d'exécution du script. Notez que le joker *, comme dans le terminal, capture le contenu du répertoire courant.

Un autre exemple plus complexe ...

Exercice : les années bissextiles

Écrire un script qui :

- Demande deux années à l'utilisateur : celle de départ ou celle d'arrivée. Elles pourront être fournies en paramètres (dans cet ordre). La ou les valeurs manquantes seront demandées par le prompt lors de l'exécution du programme le cas échéant.
- Affiche toutes les années bissextiles comprises entre celles données (incluses)

On utilisera uniquement des boucles `for` (pas de boucle `while`) !

Rappel

Une année est bissextile si elle est un multiple de 4. Exception : elle ne l'est pas si elle est un multiple de 100. Exception à l'exception : elle l'est si elle est un multiple de 400.

Indices

Indices

- Pour générer les valeurs pour la boucle `for`, allez voir du côté de ~~chez Swann~~ `seq`.
- La commande `let` ne renvoie pas directement la valeur du calcul. On utilisera une autre syntaxe pour faire les calculs dans les if : `$((calcul))`.

Solution

Proposition de solution :

```
#!/bin/bash

depart=$1
arrivee=$2

if [ -z $2 ]
then
    if [ -z $1 ]
    then
        read -p "Année de départ ? " depart
    fi
    read -p "Année d'arrivée ? " arrivee
fi

for annee in `seq $depart $arrivee`
do
    if [ $(($annee % 4)) -eq 0 ] && [ $(($annee % 100)) -ne 0 ] || [ $(($annee % 400)) -eq 0 ]
    then
        echo $annee
    fi
done
```

Retour sur les paramètres

Contexte

Tout à l'heure, on a vu que les paramètres n'étaient accessibles que via 9 variables. Mais il est bien possible d'avoir plus de 9 paramètres sur un script. On utilisera alors la commande `shift` pour « décaler » les paramètres. (`$2` → `$1`, `$3` → `$2` `[...]` → `$9`).

Exemple : écrire un script qui liste tous les paramètres avec leur place :

```
#!/bin/bash

for i in `seq 1 $#`
do
    echo "Le paramètre $i vaut $1"
    shift
done
```

Contexte

Grâce aux boucles, on sait désormais répéter plusieurs fois le même bout de code à la suite.
Maintenant, comment faire si l'on veut pouvoir répéter une suite d'instructions plusieurs fois, à des endroits différents du programme ?
Comment mieux organiser son code pour le rendre plus lisible, en créant des blocs réalisant une fonctionnalité particulière ?

Définition

Une fonction, c'est un ensemble d'instructions réutilisable permettant d'effectuer une tâche spécifique. Les fonctions peuvent prendre des **paramètres** en entrée, ce qui leur permet d'être **dynamiques**.

Déclarer une fonction

Pour déclarer une fonction, on peut utiliser deux syntaxes différentes :

```
ma_fonction()
{
    instructions
}

# OU

function ma_fonction
{
    instructions
}
```

Appeler une fonction

Explications

Après avoir déclaré une fonction, on peut l'appeler depuis n'importe où dans le script. Il suffit pour cela d'écrire le nom de la fonction.

Paramètres

Une fonction se comporte comme un mini script local. On peut lui passer des paramètres et les récupérer avec la même syntaxe que pour le programme principal.

Exemple : le script ci-dessous affichera « toto » :

```
afficheParam()
{
    echo $1
}

afficheParam toto
```

Portée des variables

Portée par défaut

Par défaut, toutes les variables ont une portée **globale**, c'est à dire qu'elles sont accessibles depuis n'importe quelle partie du script.

Portée locale

Pour faciliter le déboggage et la lisibilité d'un programme, il est conseillé d'utiliser le plus possibles de variables **locales** à l'intérieur de nos fonctions, grâce au mot-clé `local`.

```
ma_fonction()
{
    local var_locale="C'est une variable locale"
}

ma_fonction
echo $var_locale
```

Ceci n'affichera rien car la variable `$var_locale` n'est pas accessible en dehors de `ma_fonction`.

Retour de fonctions

Contexte

En Bash, on ne peut pas comme dans la plupart des autres langages renvoyer n'importe quoi depuis une fonction via un `return`. Cette directive ne permet en effet de renvoyer que des entiers (en général des codes d'erreur).

Comment faire ?

Pour récupérer la sortie d'une fonction, ou d'un programme, on utilisera la **substitution** de commande. Rappelez vous, c'est la syntaxe avec les backquotes qui permet de capturer la sortie d'une commande dans une variable ! En revanche, pour récupérer la valeur de retour d'une fonction, ou d'un programme, on utilisera la variable spéciale `$?`.

Retour de fonctions

Exemple

```
#!/bin/bash

fonction()
{
    echo 'Salut !'
    return 0
}
var=`fonction`
echo $?
echo $var
```

L'exécution affichera le résultat suivant. Comprenez-vous pourquoi ?

```
0
Salut !
```

Valeur du code de retour

Par convention, un code de retour de 0 indique le **succès** de la fonction ou du programme. C'est notamment ce qui est utilisé par Bash pour vérifier qu'une commande a réussi.

Un peu d'arithmétique

Gare à l'integer overflow...

Énoncé

Créer un script qui prend un nombre en saisie et l'élève à sa propre puissance. On vous impose d'utiliser la boucle `for`. Exemple : une entrée de 3 me renverra 3^3, soit 27.

Indices

- Pensez à l'usage de `seq`...
- Utilisez `let` pour les calculs arithmétiques.
- Il n'est pas facile de vérifier qu'une entrée est un nombre ; pas la peine d'y passer du temps.

Happy new year !

Découvrez la synthèse vocale sur Linux

Rendez-vous sur cette page !

N'hésitez pas à consulter le reste des exercices (sur la ligne de commande, pas les scripts) si vous avez du temps, après les exercices de scripts ou chez vous. Il y a même un super exercice d'édition de vidéo en ligne de commandes ;)

Vérifier l'existence d'un utilisateur

Et on rempile toute les notions !

Énoncé

Créer un script qui vaut propose le menu suivant :

- 1 - Vérifier l'existence d'un utilisateur
- 2 - Afficher le nom de tous les utilisateurs
- q - Quitter

Indices

- La structure `case` est très pratique pour vérifier un choix de l'utilisateur.
- Pensez à `grep`, `cut`, `sort`...
- Révisez le format du fichier `/etc/passwd`
- Utilisez la structure `cat /etc/passwd | while read line`...