Écrire des scripts shell

Table des matières

Contexte

Durée : 2h

Environnement de travail : Linux en ligne de commande

Pré-requis : Utiliser un terminal Linux

La ligne de commande est très utile, mais il est parfois nécessaire de réaliser des tâches complexes qui enchaînent de nombreuses commandes. Pour cela on peut écrire des scripts, créer des boucles, des conditions, etc. Ce cours va aborder les différents mécaniques qui permettent la réalisation de script sous Linux.

Écrire un script

Objectifs

- Savoir ce qu'est un script
- Savoir ce qu'est un *shell*
- Savoir écrire un script simple

Mise en situation

Il est fréquent que l'on fasse plusieurs fois les mêmes opérations sur son ordinateur. Par exemple renommer et classer dans un dossier toutes les photos produites lors d'une sortie photographie, ou effectuer des copies régulière, de sauvegarde, d'un fichier important. Écrire toutes les commandes à chaque fois est un peu ennuyeux, et surtout source de potentielles erreurs. Pour cela, on préfère utiliser des scripts.

Script ***Définition***

Un script est un fichier qui va contenir différentes instructions à exécuter dans la console, les unes à la suite des autres. De la même manière que le script d'un film décrit l'enchaînement des scènes et des répliques, un script informatique décrit les commandes successives à exécuter.

Créer un script ***Méthode***

Pour créer un script, il suffit d'écrire les commandes que l'on souhaite dans un fichier. Par exemple dans un fichier `premierscript.sh` on peut écrire une première instruction à exécuter.

```
1 echo "Hello World"
```

Pour que notre script soit utilisable, il est nécessaire de donner les droits d'exécution sur le fichier.

```
1 $ chmod +x premierscript.sh
```

Ensuite on peut lancer le script dans la console, comme si c'était une commande, en indiquant le chemin de celui-ci.

```
1 kyane@europa:~$ ./premierscript.sh
2 Hello World
```

Conseil

Lorsque l'on écrit des scripts sous Linux, la convention veut que l'on utilise l'extension `.sh` (ou bien aucune extension) pour bien identifier que c'est un fichier visant à être exécuté. Ce n'est qu'une convention, et non une contrainte technique.

Shell

En réalité lorsque l'on utilise notre ligne de commande, on utilise ce que l'on appelle un *shell*. Le *shell* est une sorte de langage de programmation : celui qui interprète les commandes qui sont entrées dans la console et qui se charge ensuite de lancer les différents programme que l'on appelle. Par défaut sous Ubuntu le *shell* est Bash (pour *Bourne-Again shell*), et c'est sans doute l'un des plus connus (il en existe d'autres, comme *sh* ou *zsh*).

Le shebang

Comme pour les instructions entrées dans la console, le script doit-être interprété par un *shell*. C'est le *shell* par défaut qui est utilisé (Bash sous Ubuntu), mais on pourrait vouloir le faire exécuter par un autre *shell*, ou même par un langage de programmation complètement différent (comme Python ou Perl).

Il existe donc une convention qui est d'indiquer en tout début de fichier de script le *shell* ou langage que l'on souhaite utiliser pour exécuter le script. C'est ce que l'on appelle un *shebang* et il a la forme suivante :

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

Un *shebang* commence systématiquement par `#!` suivi du chemin vers le *shell* que l'on souhaite utiliser (pour Bash il s'agit de `/bin/bash`). Il doit toujours être placé sur la première ligne du script.

La plupart du temps cette ligne sera toujours `#!/bin/bash` (en tout cas pour les scripts de ce cours) et n'est pas obligatoire, mais il est important de savoir ce que cela signifie.

À retenir

Si l'on écrit des commandes dans un fichier texte et que l'on ajoute des permissions d'exécution dessus, on obtient un script. Un script est pratique pour lancer une suite d'opérations identiques plusieurs fois, sans avoir à écrire les commandes à chaque fois. Le script peut contenir toutes les commandes et opérateurs que l'on écrit habituellement dans la console, car il est exécuté par le même programme (que l'on nomme le *shell*)

Exercice : Appliquer la notion

On souhaite écrire un script qui sera capable de copier tout les fichiers `.png` se trouvant dans le dossier `camera` vers un dossier `photos`, et de donner les droits en lecture/écriture uniquement au propriétaire des nouveaux fichiers. Enfin le script liste les fichiers en question pour afficher les permissions.

Pour initialiser un environnement et tester le script, vous pouvez lancer les commandes suivantes :

```
mkdir camera photos
touch camera/pic1.png
touch camera/pic2.png
touch camera/pic3.png
touch camera/pic4.png
touch camera/vid1.mp4
touch camera/vid2.mp4
touch camera/vid3.mp4
touch camera/vid4.mp4
```

Question

Écrire le script qui permet de réaliser ces opérations.

Variables et paramètres

Objectifs

- Savoir utiliser une variable en Bash
- Savoir passer un paramètre à un script

Variable

Définition

Une variable est un emplacement mémoire utilisé dans un programme ou un script pour conserver une valeur et pouvoir l'utiliser par la suite. Par exemple dans un programme qui gère un formulaire d'inscription, on peut s'attendre à trouver une variable nommée "`nom`" qui contiendrait le nom de la personne pour pouvoir l'afficher sur une interface.

Méthode

Que ce soit dans un script ou directement dans la console, il est possible de stocker des valeurs dans des variables.

```
1 $ nom="bayart"
```

Cette instruction créé simplement un variable nommée "`nom`" et qui a pour valeur "`bayart`". On peut ensuite utiliser cette variable dans les commandes suivantes, par exemple l'afficher avec la commande `echo`.

```
1 $ echo $nom
```

La commande `echo` permet d'afficher du texte ou une variable dans le terminal. Ici on fait appel à la variable "`nom`", tout simplement à l'aide du symbole "`$`" suivi du nom de la variable.

Résultat d'une commande

Méthode

Il est possible de récupérer le résultat d'une commande dans une variable. Pour cela il suffit d'entour la commande avec un `$()`.

```
1 $ resultat=$(ls -l script.sh)
2 $ echo $resultat
3 -rwxr-xr-x 1 kyane kyane 69 9 déc. 16:50 script.sh
```

Le résultat de la commande `ls -l script.sh` se trouve dans la variable `resultat` que l'on peut ensuite utiliser.

Paramètre de script

Définition

Un paramètre de script est une variable qui est passée en paramètre au lancement du script, et qui sera disponible durant son execution.

Méthode

Lorsque l'on lance un script, on peut ajouter des paramètres, séparés par des espaces, qui seront utilisables dans le script. Par exemple :

```
$ ./hello.sh kyane pichou
```

Ici on lance le script `hello.sh` qui va recevoir deux paramètres : `kyane` et `pichou`. On peut les utiliser dans le script de la manière suivante :

```
#!/bin/bash

echo "Bonjour $1 $2"
```

```
$ ./hello.sh kyane pichou
Bonjour kyane pichou
```

Les paramètres se trouvent dans des variables ayant pour nom des chiffres. Le premier paramètre est dans `$1`, le second dans `$2`, etc. Ici on affiche donc le texte "Bonjour" suivi des deux paramètres.

À retenir

Il est possible de créer des variables, en console on dans un script, qui vont conserver en mémoire une valeur ou le retour d'une commande, pour la réutiliser plus tard. De plus il est possible d'accéder à des variables, dans un script, contenant les paramètres passés lors de l'exécution

Exercice : Appliquer la notion

On souhaite améliorer le script suivant.

```
#!/bin/bash

cp camera/*.png photos/
chmod 600 photos/*
ls -l photos
```

Ce script copie tout les fichiers `.png` se trouvant dans le dossier `camera` vers un dossier `photos`, change les droits en lecture/écriture uniquement au propriétaire, puis liste les fichiers pour afficher les permissions.

L'objectif est de le modifier pour qu'il n'affiche plus la liste des permissions des nouveaux fichiers, mais qu'il réalise 2 choses :

- il prend le nom du dossier de destination en paramètre du script
- il écrit, à la fin, un message indiquant "Il y a X photos dans le dossier" (où X est le nombre de fichiers `.png` dans le dossier de destination)

Pour initialiser un environnement et tester le script, vous pouvez lancer les commandes suivantes :

```
mkdir camera photos
touch camera/pic1.png
touch camera/pic2.png
touch camera/pic3.png
touch camera/pic4.png
touch camera/vid1.mp4
touch camera/vid2.mp4
touch camera/vid3.mp4
touch camera/vid4.mp4
```

Question

Modifiez le script en conséquence.

Structures conditionnelles

Objectifs

- Comprendre la notion de structure conditionnelle
- Savoir faire un `if` en Bash
- Savoir utiliser `elif` et `else`

Structure conditionnelle

Définition

En programmation, une structure conditionnelle est une structure qui permet, dans un script ou un programme, d'exécuter des instructions en fonction de conditions basées sur les valeurs de variables.

Exemple

Voici un exemple rédigé (on parle d'un algorithme) d'une structure conditionnelle.

```
SI "mavariable" est plus grande que "12"
ALORS
  Afficher un message d'erreur
FIN SI
```

On comprends qu'une condition (*« si mavariable est plus grande que 12 »*) est évaluée, et si cette condition est validée alors on exécute l'instruction dans la structure (*« Afficher un message d'erreur »*).

Syntaxe

La même structure en Bash donne :

```bash
if [ $mavariable -gt 12 ]
then
  echo "Error"
fi
```

La partie se trouvant entre les crochets est donc la condition. On retrouve bien l'appel à `mavariable` (avec le symbole `"$"`), que l'on compare à la valeur 12. La comparaison se fait avec l'opérateur `-gt`, signifiant *Greater Than*, qui permet de valider que ce qui se trouve à gauche de l'opérateur est plus grand que ce qui se trouve à droite.

Attention

Une condition se trouve systématiquement entre crochets, avec un espace entre chaque crochet et la condition. Par exemple écrire `[$mavariable -gt 12]` n'est pas valide.

Les différentes conditions

Syntaxe

Il existe de nombreux opérateur pour les conditions, voici quelques exemples pour les comparaison de variables contenant du texte.

`$texte1 = $texte2`	Vérifie que la variable `texte1` contient la même chose que `texte2`
`$texte1 != $texte2`	Vérifie que les deux variables sont différentes
`-z $texte`	Vérifie si la variable est vide (pas de texte) ou inexistante.
`-n $texte`	Vérifie si la variable est non vide

D'autres permettent la comparaison de nombres.

`$num1 -eq $num2`	Vérifie que les 2 nombres sont égaux
`$num1 -ne $num2`	Vérifie que les 2 nombres ne sont pas égaux
`$num1 -lt $num2`	Vérifie si `num1` est inférieur (<) à `num2`
`$num1 -le $num2`	Vérifie si `num1` est inférieur ou égal (<=) à `num2`
`$num1 -gt $num2`	Vérifie si `num1` est supérieur (>) à `num2`
`$num1 -ge $num2`	Vérifie si `num1` est supérieur ou égal (>=) à `num2`

Enfin certains permettent de faire des tests sur des fichiers.

`-e $nomfichier`	Vérifie si le fichier existe
`-d $nomfichier`	Vérifie que le fichier est un répertoire.
`-f $nomfichier`	Vérifier que le fichier est un fichier (donc pas un répertoire)

Inversion de condition

Méthode

Il est possible d'inverser une condition en utilisant la caractère de négation "`!`".

```
1 if [ ! $mavariable -gt 12 ]
2 then
3   echo "Error"
4 fi
```

Ici l'instruction dans le bloc `if` ne sera exécutée que si `mavariable` **n'est pas** plus grande que 12. La condition est inversée.

Sinon

Méthode

Il est possible de proposer une alternative si la condition première n'est pas respectée.

```
1 if [ $mavariable -gt 12 ]
2 then
3   echo "Error"
4 else
5   echo "Ok"
6 fi
```

Si la condition est validée on exécute l'instruction dans le bloc `if`, sinon on exécute celle du bloc `else`.

Sinon si

Méthode

Il est possible de complexifier les conditions en ajoutant plusieurs alternatives conditionnelles avec le mot clef `elif` (contraction de *else if*).

```
if [ $mavariable -gt 12 ]
then
  echo "Error"
elif [ $mavariable -lt 5 ]
then
  echo "Perfect"
else
  echo "Ok"
fi

```

Si la variable est plus grande que 12, alors on affiche "Error", sinon si elle est plus petite que 5 on affiche "Perfect", sinon on affiche "Ok".

À retenir

Les conditions permettent de réaliser différentes instructions de manière conditionnelle. En fonction de la valeur de variables, on peut décider ou non d'exécuter une instruction, et de définir une ou plusieurs alternatives si la condition n'est pas respectée.

Exercice : Appliquer la notion

On souhaite améliorer le script suivant.

```
#!/bin/bash

cp camera/*.png $1/
count=$(ls $1 | wc -l)
echo "Il y a $count photos dans le dossier"
```

Ce script copie tout les fichiers `.png` se trouvant dans le dossier `camera` vers un dossier de destination, puis compte le nombre de fichiers dans le dossier de destination pour l'afficher. Le dossier de destination est déterminé pour le premier argument envoyé au script.

```
$ ./script.sh photos
```

Question 1

Que se passe-t-il si l'on oublie de donner le dossier de destination en argument au lancement du script ?

Question 2

À l'aide des conditions, adaptez le script pour vérifier que l'argument a bien été donné au script, et sinon utilisez un dossier de destination par défaut nommé `photos`.

Les opérateurs logiques

Objectifs

- Comprendre la notion d'opérateur logique
- Savoir utiliser des opérateurs OR et AND en Bash

Opérateurs logiques

La plupart des langages de programmation permettent de complexifier les conditions à l'aide d'opérateurs logiques. Les opérateurs logiques permettent de combiner deux conditions pour en former une nouvelle, plus complexe. Par exemple la condition "SI ceci ET cela" est formée de deux conditions ("ceci" et "cela") qui sont combinées à l'aide d'un opérateur logique qui est "ET".

Opérateur ET (&&) *Méthode*

Il est possible d'assembler 2 conditions entre-elles pour former condition plus complexe qui n'est valide que si les deux sous-conditions sont vérifiées. C'est l'opérateur ET qui s'écrit `&&` .

```
if [ $prenom = "aaron" ] && [ $nom = "swartz" ]
then
  echo "Bonjour Aaron Swartz"
fi
```

L'instruction sera exécutée que si les 2 conditions sont validées

Opérateur OU *Méthode*

L'opérateur OU, qui s'écrit `||`, permet de faire la même chose que `&&` mais seulement si au moins une des conditions est correcte.

```
if [ $jour = "samedi" ] || [ $jour = "dimanche" ]
then
  echo "C'est le week-end"
fi
```

L'instruction sera exécutée si au la variable `jour` a pour valeur "samedi" ou "dimanche".

À retenir

Les opérateurs logiques permettent de former des conditions plus complexe, et donc de réaliser des algorithmes plus poussés.

Exercice : Appliquer la notion

On souhaite améliorer le script suivant.

```
1 #!/bin/bash
2
3 cp camera/*.png $1/
4 count=$(ls $1 | wc -l)
5 echo "Il y a $count photos dans le dossier"
```

Ce script copie tout les fichiers `.png` se trouvant dans le dossier `camera` vers un dossier de destination, puis compte le nombre de fichiers dans le dossier de destination pour l'afficher. Le dossier de destination est déterminé pour le premier argument envoyé au script.

```
1 $ ./script.sh photos
```

On souhaite que le script prenne désormais en premier paramètre le dossier de départ et en second paramètre le dossier de destination. Si l'un des deux paramètres n'est pas bon, il doit écrire un message d'erreur.

Question

À l'aide de conditions et d'opérateurs logiques, écrire le script amélioré.

Les boucles

Objectifs

- Comprendre la notion de boucle
- Savoir faire un `while`
- Savoir faire un `for`

Boucle *Définition*

En programmation une boucle est une structure qui permet de répéter plusieurs fois un même bloc d'instructions.

Boucle while

La boucle `while` (signifiant "tant que" en anglais) permet de répéter un bloc d'instruction tant qu'une condition est remplie. Elle se base sur les mêmes types de conditions que les structures conditionnelles.

Syntaxe

Pour écrire cette boucle, il suffit d'utiliser le mot clef `while` suivi des conditions à remplir pour continuer la boucle. Le bloc d'instructions est délimité par un `do` en début et un `done` à la fin.

```
1 while [ -z $reponse ] || [ $reponse != 'bonjour' ]
2 do
3   read -p 'Dites "bonjour" : ' reponse
4 done
```

La commande `read` permet de demander à l'utilisateur de saisir une valeur au clavier, et met cette valeur dans la variable `reponse`. La boucle `while` va demander à l'utilisateur de rentrer une valeur tant que la variable `reponse` n'est pas définie ou qu'elle ne contient pas le mot "bonjour".

Boucle for

La boucle `while` est intéressante lorsque l'on ne sait pas à l'avance combien de fois on va refaire la boucle (on parle d'itérations). Cependant dans certains cas on sait à l'avance combien de tours de boucle on souhaite faire, par exemple si l'on souhaite appliquer un même traitement à une liste de variables. Dans ce cas, on utilise la boucle `for`.

Syntaxe

La syntaxe de la boucle `for` est la suivante :

```
1 for ELEMENT in LISTE
2 do
3   INSTRUCTIONS
4 done
```

La `LISTE` est une successions de valeurs sur lesquelles on va itérer. À chaque tour de boucle, une variable nommée `ELEMENT` va être utilisable dans tout le bloc d'`INSTRUCTIONS`. Une fois que chaque élément de la liste a été traité, la boule s'arrête.

Boucle sur une variable

Exemple

Cette boucle `for` permet de saluer différentes personnes à la suite.

```
for nom in "Kyâne" "Stéphane" "Quentin"
do
  echo "Bonjour $nom"
done

```

```
$ ./script.sh
Bonjour Kyâne
Bonjour Stéphane
Bonjour Quentin
```

Dans Bash les listes sont généralement des éléments séparés par des espaces. De ce fait il est courant de mettre la liste des éléments dans une variable et de l'utiliser dans l'instruction `for`.

```
personnes="Kyâne Stéphane Quentin"
for nom in $personnes
do
  echo "Bonjour $nom"
done

```

Boucle sur une commande

Exemple

Il est aussi possible d'utiliser la boucle `for` pour faire un nombre précis d'itérations.

```
for i in {1..10}
do
    echo "$i"
done
```

Utiliser `{1..10}` permet d'indiquer à l'instruction `for` qu'il faut itérer sur tout les nombres de 1 à 10 (inclus). Dans ce cas la variable (ici `i`) va contenir le nombre auquel on se trouve dans la boucle.

À retenir

Les boucles permettent de réaliser plusieurs fois le même bloc d'instructions. La boucle `while` est pratique pour boucler tant qu'une condition n'est pas remplie, tandis que la boucle `for` est utilisé pour exécuter une instruction un nombre défini de fois.

Exercice : Appliquer la notion

On souhaite améliorer le script suivant. Ce script prends en paramètre un nom de dossier, copie tout les fichiers `.png` se trouvant dans le dossier `camera` vers le dossier passé en paramètre, puis compte le nombre de fichiers dans le dossier de destination pour l'afficher.

```
#!/bin/bash

if [ -z $1 ]
then
  dossier="photos"
else
  dossier=$1
fi

cp camera/*.png $dossier/
count=$(ls $dossier | wc -l)
echo "Il y a $count photos dans le dossier"
```

On souhaite maintenant que ce script ne copie pas tout les fichiers `.png` d'un coup, mais qu'il réalise la copie un par un tout en affichant un message "Copie de CHEMIN_DU_FICHIER" à chaque étape.

Question

Adaptez le script en conséquence.

Les fonctions

Objectifs

- Comprendre la notion de fonction
- Savoir définir une fonction
- Savoir découper son programme

Fonction

 *Définition*

Une fonction est un ensemble d'instructions qui permettent d'effectuer une tâche particulière, elle peut prendre des paramètres

On peut voir une fonction comme un sous script.

§ *Syntaxe*

Pour définir une fonction il existe deux syntaxe en Bash. L'une comme l'autre permettent de faire exactement la même chose mais chacun peut avoir ses préférences notamment en terme de lisibilité du code.

```
1 function mafonction {
2     # Un traitement
3 }
4
5 # OU
6
7 mafonction() {
8     # Un traitement
9 }
```

Une fois qu'une des deux syntaxe a été adoptée, une bonne pratique est de garder la même syntaxe tout le long du projet afin de garder une certaine cohérence.

Appeler une fonction

§ *Syntaxe*

On peut simplement appeler une fonction quand elle a été définie. On utilise une syntaxe similaire à l'appel des commandes, à savoir le nom de la fonction éventuellement suivi de ses paramètres.

```
1 function bonjour {
2     echo "Bonjour $(whoami)
3 }
4
5 bonjour
```

Utiliser des paramètres

Syntaxe

On a vu qu'il était possible d'utiliser des paramètres. Tout comme pour un script, les paramètres sont accessibles à l'aide de variables spéciales qui ont pour nom `1`, `2`, `3`, ... On y accède donc avec `$1`, `$2`, `$3`, ...

```
1 function square {
2     echo "Le carre de $1 est $(($1*$1))"
3 }
4
5 square 3
```

Récupérer un résultat

Syntaxe

Pour récupérer le résultat de l'exécution d'une fonction on pourra utiliser la sortie standard de cette fonction tout comme pour récupérer le résultat d'une commande.

```
1 function square {
2     echo $(($1*$1))
3 }
4
5 resultat=$(square 3)
6 echo "Le carre de 3 est $resultat"
```

Quand utiliser les fonctions ?

Méthode

Les fonctions sont très utiles pour découper de manière logique le code, par exemple si vous voyez qu'une suite d'instructions similaire est souvent répétée dans votre code ou si à un endroit vous utilisez une séquence d'instruction qui a pour but d'effectuer une instruction bien précise (interpréter un texte par exemple).

Fondamental

Les fonctions permettent d'éviter de répéter une séquence d'instruction à plusieurs endroits (rappelez vous, un bon programmeur est un programmeur fainéant !). Elles permettent également de modifier le comportement d'une séquence d'instructions à un seul endroit, sans avoir à se rappeler de tous les endroits où elle est utilisée, on gagne ainsi du temps et on évite les oublis.

Exercice : Appliquer la notion

On souhaite améliorer le script suivant. Ce script prends en paramètre un nom de dossier, réalise la copie un par un des fichiers du dossier tout en affichant un message "Copie de CHEMIN_DU_FICHIER" à chaque étape.

```
#!/bin/bash

if [ -z $1 ]
then
  dossier="photos"
else
  dossier=$1
fi

for fichier in $(ls camera/*.png)
do
  echo "Copie de $fichier"
  cp $fichier $dossier/
done

count=$(ls $dossier | wc -l)
echo "Il y a $count photos dans le dossier"
```

On souhaite maintenant que ce script utilise une fonction `copie_fichier` qui s'occupe de faire le traitement de la copie du fichier pour chaque fichier. Cette fonction doit fonctionner peut importe le fichier source et le dossier destination.

Question

Adaptez le script en conséquence.

Quiz

Exercice 1 : Quiz - Culture

Exercice

Dans un script les boucles permettent

- [ ] de répéter une action tant qu'une condition reste vraie
- [ ] de renvoyer le résultat d'une commande à une autre commande
- [ ] de répéter une même action sur tout les éléments d'une liste
- [ ] de lier plusieurs variables entre elles

Exercice

Quel est la bonne syntaxe si l'on souhaite lancer un script nommé "calcul.sh" avec pour paramètres `4` et `12`

- ( ) PARAMS=4,12 ./calcul.sh
- ( ) ./calcul.sh --params 4 12
- ( ) ./calcul.sh 4 12
- ( ) ./calcul.sh $1=4 $2=12

Exercice

Comment est-il possible de faire référence à une variable `mavar` en Bash ?

- ( ) `!mavar`
- ( ) `mavar`
- ( ) `$mavar`
- ( ) `^mavar`
- ( ) `(mavar)`

Exercice

Une fonction

- ☐ est utilisée par le script pour calculer l'image `y` d'un point `x` à l'aide d'une expression mathématique.
- ☐ permet de factoriser des instructions ou des comportements qui reviennent souvent dans le programme.
- ☐ peut prendre des paramètres.
- ☐ doit prendre des paramètres.

Exercice 6 : Quiz - Méthode

Exercice

Quels types de structures conditionnelles existent en Bash ?

- ☐ SI...
- ☐ PEUT-ÊTRE...
- ☐ SI...SINON SI...SINON
- ☐ SI...AUSSI SI...

Exercice

Quels types d'opérateurs logiques existent en Bash ?

- ☐ OU EXCLUSIF
- ☐ ET
- ☐ OU
- ☐ ET EXCLUSIF

Exercice

Pour créer un script il faut :

- ○ Utiliser la commande `makescript` puis écrire dans le fichier créé
- ○ Écrire des instructions dans un fichier et le rendre exécutable
- ○ Utiliser un logiciel d'édition de scripts, comme VisualStudio Code ou Atom.

Exercice 10 : Quiz - Code

Exercice

Si l'on écrit ceci dans le terminal : `./script 4`

- [ ] Cela va exécuter le fichier nommé `script`
- [ ] Cela va créer 4 fichiers nommés `script1`, `script2`, `script3`, `script4`
- [ ] La variable `$1` dans le script contiendra la valeur 4
- [ ] La variable `$param1` dans le script contiendra la valeur 4
- [ ] Une erreur indiquant que la commande `script` n'existe pas en Bash va s'afficher.

Exercice

Que va afficher le script suivant ?

```bash
#!/bin/bash

nb=18
if [ $nb -lt 18 ] || [ ! -z $nb]
then
  echo "Score insuffisant"
else
  echo "Bravo"
fi
```

- ○ "Score insuffisant"
- ○ "Bravo"
- ○ Rien, il y a une erreur le script ne va pas fonctionner.

Exercice

Que va afficher le script suivant ?

```bash
#!/bin/bash

for nb in {4..7}
do
  if [ $nb -ge 6 ]
  then
    echo $nb
  fi
done
```

- ○ 4
  5
  6
  7
- ○ 6
- ○ 6
  7
- ○ 5
  6

○ 7