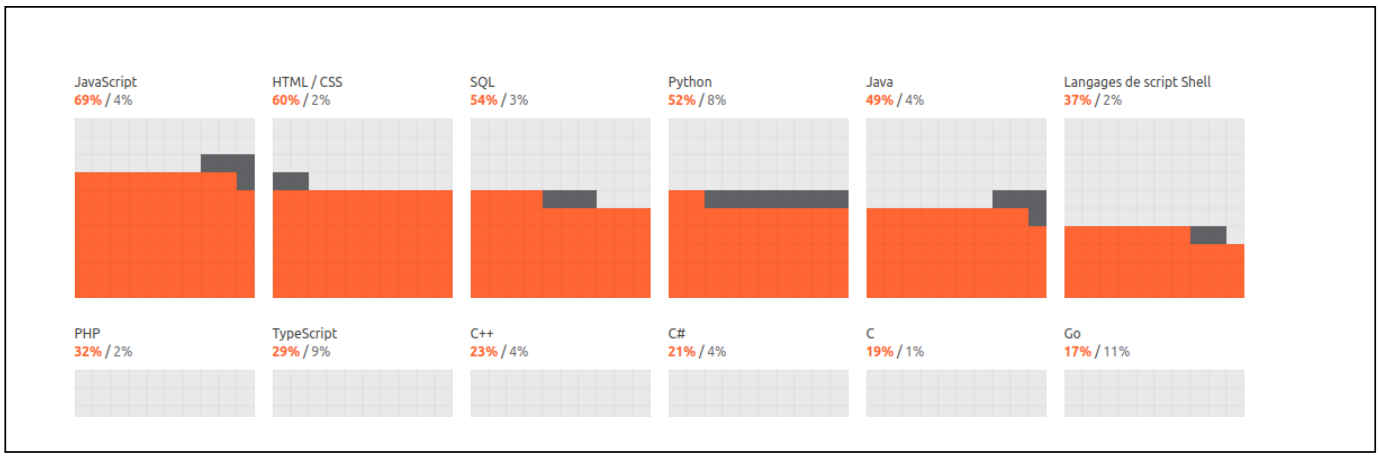
1 Écriture d'un script shell

Utilisation du BASH



doc jetbrains

- **SCRIPT** : code qui contient une série de commandes.
  - Ces commandes sont exécutées par un **interpréteur** les unes après les autres (le shell pour les scripts que nous allons voir).
- Le scripting est la méthode idéale pour l'automatisation de tâches et notamment la mouvance DevOPS

1.1 shebang

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

```
#!<chemin-vers-interpreter>
#!/bin/sh
#!/bin/csh
#!/bin/zsh
#!/usr/bin/python
#!/c/Windows/System32/WindowsPowerShell/v1.0/powershell.exe -File
```

Sans Shebang au début du script, les commandes à l'intérieur du script seront exécutées en utilisant votre propre shell

Outrepasser le shebang : `bash mon-script.sh`

1.2 premier script

```
echo $SHELL
touch exo1.sh
chmod u+x exo1.sh

# exécution du script
./exo1.sh
```

Il est important de mettre les droits sur le fichier contenant le script de manière à ce que celui-ci devienne exécutable

Tester la commande :

```
env
```

Rechercher la variable d'environnement `SHELL` pour connaître le shell par défaut (faire un essai en ajoutant derrière la commande `| grep SHELL`)

1.3 les variables

Pour utiliser les variables et afficher le contenu associé, il faut faire précéder le nom de la variable par un $

```
#!/bin/bash
IUT="Belfort/Montbéliard"
BUT="Informatique"
DUREE=3
echo "J'étudie à l'IUT de $IUT pour obtenir un Bachelor Universitaire Technologique en $BUT en ${DUREE}an(s)"
```

Enlever les `{}` dans l'instruction `echo`

- Les variables ne peuvent pas commencer par un chiffre mais peuvent en contenir
- Les variables ne peuvent pas contenir de tiret (-)
- Les variables ne peuvent contenir que des underscores, majuscules, minuscules et chiffres

```
TEST="monTest"
TEST1=${TEST}
echo $TEST1
```

```
MACHINE=${HOSTNAME}
MACHINE2=$(hostname)
echo "MACHINE ${MACHINE} MACHINE2 ${MACHINE2} "
```

beaucoup plus d'informations

Vu en TD:

`$?` code de retour de la dernière commande. Vaut généralement 0 si cette commande s'est bien déroulée, et un autre nombre correspondant à un type d'erreur, décrit par la commande `errno` (`sudo apt install moreutils`)

Exercice 1

Réaliser un script shell qui permet de :

- mettre dans une variable `varpy2` le contenu de la Sortie standard de la commande `python2 --version`
- mettre dans une variable `retComPy2` la valeur de retour de la commande `python2 --version`
- mettre dans une variable `varpy3` le contenu de la Sortie standard de la commande `python3 --version`
- mettre dans une variable `retComPy3` la valeur de retour de la commande `python3 --version`
- mettre dans une variable `varpy` le contenu de la Sortie standard de la commande `python --version`
- mettre dans une variable `retComPy` le retour de la commande `python --version`

Pour interpréter et récupérer le résultat d'une commande (contenu de la Sortie standard), utiliser la syntaxe `var=$(commande)`

REMARQUE :
La valeur de retour d'une commande se trouve dans la variable `$?` ; Si ce retour vaut `0` c'est que la commande a répondu correctement.
Si la valeur de retour est différente de `0` , le contenu de **la Sortie standard** est vide, dans le terminal c'est le contenu de **la Sortie d'erreurs** qui est affiché (voir TD)
Le contenu de la Sortie standard de la commande sera vide si la commande n'est pas connu (un message d'erreur est affiché sur la sortie d'erreur standard) et le retour sera `127` (commande inconnue)

- Afficher le contenu de toutes ces variables

- Tester la commande `declare -p` après exécution du script

1.4 les Tests

```
man test
test  EXPRESSION
[ EXPRESSION ]
```

`[ voici-la-condition-du-test-a-verifier ]` : Il est important de respecter les espaces après le `[` mais également avant le `]`

- Parmi les opérateurs principaux nous avons :
  - `-e` : 0 (True) si le fichier existe **sinon 1 (False) sinon 2 (error)**
  - `-d` : 0 (True) s'il s'agit d'un **dossier**
  - tester les droits
    - `-r` : 0 (True) si le fichier est disponible en lecture pour l'utilisateur
    - `-w` : 0 (True) si le fichier est disponible en écriture pour l'utilisateur
    - `-x` : 0 (True) si le fichier est disponible en exécution pour l'utilisateur
  - `-s` : 0 (True) si le fichier existe et n'est pas vide
  - les chaînes de caractères
    - `-z` : 0 (True) si la chaîne de caractères est vide
    - chaine1 `=` chaine2 : 0 si chaine1 est égale à chaine2
    - chaine1 `!=` chaine2: 0 si chaine1 est différente de chaine2
  - les numériques
    - chiffre1 `-eq` chiffre2 : 0 si chiffre1 est égal à chiffre2
    - chiffre1 `-ne` chiffre2 : 0 si chiffre1 est différent de chiffre2
    - chiffre1 `-lt` chiffre2 : 0 si chiffre1 est plus petit que chiffre2
    - chiffre1 `-le` chiffre2 : 0 si chiffre1 est plus petit ou égal que chiffre2
    - chiffre1 `-gt` chiffre2 : 0 si chiffre1 est plus grand que chiffre2
    - chiffre1 `-ge` chiffre2 : 0 si chiffre1 est plus grand ou égal que chiffre2

doc test - doc test suite

documentation sur les tests

```
[ -e /home/amillet/.bashrc ]
echo $?
TEST=""
[ -z $TEST ]
echo $?

TEST="monTest"
[ -z $TEST ]
echo $?
```

```
TEST2="monTest"
echo $TEST $TEST2
[ $TEST2 = $TEST ]  # attention aux espaces
echo $?
[ $TEST2 != $TEST ]  # mettre une chaîne vide
echo $?


CHIFFRE1=14
CHIFFRE2=18
[ $CHIFFRE1 -eq $CHIFFRE2 ]
echo $?
[ $CHIFFRE1 -ne $CHIFFRE2 ]
echo $?
```

1.5 L'utilisation du "Si Alors Sinon" IF THEN ELSE

```
if [ condition-est-vraie ]
then
    command
    command2
fi
```

```
if [ condition-est-vraie ]
then
    command
    command2
else
    command3
    command4
fi
```

- exemple

```
#!/bin/bash

if [ -e /home/tpreseau/.bashrc ]
then
    echo "Le fichier .bashrc existe bien"
else
    echo "Le fichier .bashrc n'existe pas"
fi
```

```
if [ condition-est-vraie ]
then
    command
elif [ condition-est-vraie ]
then
    command
else
    command
fi
```

```
#!/bin/bash
CHIFFRE1='16'
CHIFFRE2='17'
if [ $CHIFFRE1 –lt $CHIFFRE2 ]
then
    echo "$CHIFFRE1 est plus petit que $CHIFFRE2"
elif [ $CHIFFRE1 –gt $CHIFFRE2 ]
then
    echo "$CHIFFRE1 est plus grand que $CHIFFRE2"
else
    echo "$CHIFFRE1 est égal à $CHIFFRE2"
fi
```

Exercice 2

Réaliser un script shell qui permet de faire un petit diagnostique sur l'installation de python sur une machine Linux :

A va ré-utiliser le script précédent, à l'aide des variables, afficher si les commandes `python2` , `python3` , `python` fonctionnent ainsi que le chemin de ces commandes si elles fonctionnent (avec la commande `which` et éventuellement l'option `-a`)

Pour faire ce un petit diagnostique de python, utiliser l'algorithme ci-dessous :

- SI la valeur de retour de la commande `python2 --version` est bonne
  - afficher "Python2 installé. Le chemin du programme est ….."
  - afficher "La version de python 2 installée est …."
- SINON
  - afficher "Python2 absent. Pour installer Python 2, exécutez la commande suivante en root :"

    - afficher “apt install python2”
- FSI
- SI la valeur de retour de la commande `python3 --version` est bonne
    - afficher “Python3 installé. Le chemin du programme est …..”
    - afficher “La version de python 3 installée est ….”
- SINON
    - afficher “Python3 absent. Pour installer Python 3, exécutez la commande suivante en root :”
    - afficher “apt install python3”
- FSI
- SI la valeur de retour de la commande `python --version` est bonne
    - SI le retour (contenu de la sortie) de la commande `python2 --version` est identique à celui de `python --version`
        - afficher “La commande python est installée et renvoie sur Python 2”
    - SINON
        - SI le retour (contenu de la sortie) de la commande `python3 --version` est identique à celui de `python --version`
            - afficher “La commande python est installée et renvoie sur Python 3”
        - SINON
            - afficher “La commande python est installée, mais renvoie vers une version inconnue de python : ….”
        - FSI
    - FSI
- FSI

TEST

- tester votre script
- Installer python3
- Exécuter l’instruction pour créer un lien symbolique python sur python 3 si il n’existe pas

Que fait cette commande `ln -sf $(which python3) $(which python)` ?

Pour les plus rapides :

Tester la commande `apt install python-is-python3`

Tester et améliorer votre script avec le code ci-dessous

```
reponseCmd=$(python3 --version)
set $reponseCmd
echo $2
```

1.6 la boucle POUR (FOR IN)

```
for VARIABLE in OBJET1 OBJET2 OBJET3 OBJETn
do
    command
    command2
done
```

```
#!/bin/bash
for CHIFFRE in 10 11 12 13
do
    echo "Chiffre : $CHIFFRE"
done
```

```
#!/bin/bash
CHIFFRES="10 11 12 13"
for CHIFFRE in $CHIFFRES
do
    echo "Chiffre : $CHIFFRE"
done
```

Exercice 3

Créer un script dans un dossier composé de au moins 2 fichiers et 2 répertoires (dossiers)

La commande `ls` retourne dans une liste de valeurs.

`for fichier in *`

Indiquer pour chaque fichier si le fichier est un fichier ordinaire ou un répertoire.

1.7 les variables de positionnement

Les variables de position stockent le contenu des différents éléments de la ligne de commande utilisée pour lancer le script.

- Il en existe 10 : $0 jusqu’à $9
    - Le script lui-même est stocké dans la variable $0

- Le premier paramètre est stocké dans la variable $1
- Le second paramètre est stocké dans la variable $2
- …

créer un script `test_variables_posi.sh`

```
#!/bin/bash
echo "Voici les paramètres utilisés : $@"

echo "nom du script :  $0 ;  argument1 : $1 ; argument2 : $2"

echo "Voici les paramètres utilisés : $@"

echo "Voici le nombre de paramètres à partir de \$1 : $#"

echo "Voici les paramètres utilisés à partir de \$* : $*"

echo $9
shift
echo $9
```

`./test_variables_posi.sh a1 a2 a3 a4 a5 a6 a7 a8 a9 a10`

Tester le script ci-dessous:

```
#! /bin/bash
if [ $# -ne 1 ]
then
    echo “SYNTAXE : $0 répertoire”
else
    echo 'ls -la $1'
    echo $?
    echo "ls -la $1"
    echo $?
    echo `ls -la $1`
    echo $?
    echo $(ls -la $1)
    echo $?
fi
```

Exercice 4

écrire un script `test_exo4.sh` qui affiche tous les paramètres à l’aide d’une boucle

si l’utilisateur n’a pas saisi d’argument, afficher un message d’erreur suivi du code erreur : `exit 2`

2 code retour : erreur

A chaque fois qu’une commande est exécutée, elle renvoie un code de sortie (exit code :Entier compris entre 0 et 255)

Dans la plupart des langages de développement, une commande qui s’est exécutée correctement renvoie un code retour égal à 0, sinon c’est qu’une erreur s’est passée au moment de l’exécution du code

```
#!/bin/bash
HOTE=$1
NOMBRE_DE_PAQUETS=$2
ping -c $NOMBRE_DE_PAQUETS $HOTE
if [ "$?" -ne "0" ]
then
    echo "L'hote $HOTE n'est pas joignable"
    #exit 1
else
    echo "L'hote $HOTE est joignable"
    #exit 0
fi
```

`ping -c 1 8.8.8.8`

`exit` permet au script de quitter avec un code erreur différent de 0 (`exit 1` ou `exit 2`)

3 le ET et le OU

le ET : exécute la commande suivante à droite si le code erreur renvoyé par la commande est égal à “0”

```
ls -l test
echo $?
ls -l test && mkdir test/d1
```

```
echo $?
mkdir test && mkdir test/d1
echo $?
rmdir test/d1 && mkdir test
```

le OU : exécute la commande suivante à droite si le code erreur par la commande est différent de “0”

```
ls -l test || mkdir test
```

Utile pour la prise de décision en une ligne (SI)

4 les fonctions

```
function internet() {
    ping -c $1 $2
    if [ $? -eq 0 ]
    then
        echo "La connectivité vers internet est établie"
    else
        echo "Pas de connectivité vers internet"
    fi
}

internet "1" "8.8.8.8"
internet "1" "www.facebook.com"
internet "1" "www.mauvaise_url.fr"
```

Exercice 5

6/ Nom de la commande **test_exo5.sh**
Arguments : nom d’un fichier ordinaire non vide existant et ayant le droit de lecture
Effet : La commande doit afficher les messages suivants
“Le nom du fichier est : …”
“Le nombre de caractères est : …”
“Le nombre de mots est : …”
“Le nombre de lignes est : …”

Utiliser la commande wc

Exercice 6

4/ Nom de la commande **test_exo6.sh**
Arguments : Aucun
Effet : La commande doit afficher les messages suivants
`"Nous sommes le "numéro du jour" / "jour" / "mois" / "année""`
Remarque : Utiliser la commande : date (voit aussi `man date`)

tester le script :

```
a=$(date)
set $a
echo $1 :: $2
```

modifier ensuite la variable IFS

```
old_IFS=$IFS
IFS=${IFS}:
# reprendre le script précédent
# traitement de la date
IFS=$old_IFS
```

pour les plus rapides, mettre un peu de couleur

`echo -e '\033[1;31m' ROUGE '\033[0m'`

Code BASH :

```
# Réinitialisation de la couleur après cette balise
COLOR_RESET='\033[0m'
# Codes couleurs à placer avant le texte :
COLOR_NOIR='\033[0;30m'
COLOR_ROUGE='\033[0;31m'
COLOR_VERT='\033[0;32m'
COLOR_JAUNE='\033[0;33m'
COLOR_BLEU='\033[0;34m'
COLOR_VIOLET='\033[0;35m'
COLOR_CYAN='\033[0;36m'
COLOR_BLANC='\033[0;37m'
```

source

Mini Projet Système :

Objectif FINAL : créer une commande pour dé-archiver tous les projets des étudiants et copier dans le même dossier que le fichier app.py un fichier **test_projet.sh** ; utiliser le fichier joint en fin de tp

Premier pas pour résoudre le problème :

Ma propre commande **ls** avec extract : **ls_extract.sh**

dans le dossier `/tmp` par exemple, executer le script ci-dessous

```
mkdir tmp_exo5 ; ce tmp_exo5
mkdir detu1 detu2 detu3
touch detu1/projet1.zip detu2/projet2.tar.gz detu3/projet3.zip
touch ls_extract.sh launcher.sh
```

On souhaite créer une commande permettant de lister les fichiers d'un dossier avec un affichage "customisé", puis désarchiver ces fichiers si c'est possible.

41. La commande ls affiche par défaut les éléments sur une ligne:

```
>ls
detu1 detu2 detu3 launcher.sh  ls_extract.sh
```

Écrire un script **ls_extract.sh** permettant de lister les éléments (dossiers et fichiers) du répertoire courant en affichant un nom par ligne:

```
>./ls_extract.sh
detu1
detu2
detu3
launcher.sh
ls_extract.sh
```

(Remarque : utiliser une boucle `for`)

2. On souhaite maintenant pouvoir distinguer les dossiers des fichiers. Modifier le script pour qu'il affiche `+ rep:` devant les nom de dossiers:

```
>./ls_extract.sh
+ rep: detu1
+ rep: detu2
+ rep: detu3
launcher.sh
ls_extract.sh
```

(Remarque : faire un test avec un SI dans la une boucle `for`)

3. On voudrait également voir les fichiers contenus dans les dossiers. Modifier le script pour qu'il affiche les éléments contenus dans un dossier en les décalant d'une tabulation et indiquer le type de fichier (`.zip` ou `.tar.gz`):

```
>./ls_extract.sh
arborescence.sh
+d rep: detu1
    -projet1.zip : fichier zip
+d rep: detu2
    -projet2.tar.gz : fichier tar.gz
+d rep: detu3
    -projet1.zip : fichier zip
launcher.sh
ls_extract.sh
```

Remarque pour savoir si le fichier se termine par `.zip` on peut utiliser la commande `file=$(ls *.zip)  2> /dev/null` , puis tester le retour de la commande. Faire de même avec l'autre extension.

Autre solution (arnaud)

```
case $fichier in
*.jpg) echo "C'est un JPEG!";;
esac


if [[ $fichier == *.jpg ]]; then echo "C'est un JPEG!"; fi
```

4. l'extraction

Votre enseignant reçoit beaucoup de projets. Pour chaque projet , afficher la commande à exécuter pour extraire l'archive, exemple unzip $file ou tar xvf $file

```
>./ls_extract.sh
arborescence.sh
+d rep: detu1
    -projet1.zip : fichier zip
+d rep: detu2
    -projet2.tar.gz : fichier tar.gz
+d rep: detu3
    -projet1.zip : fichier zip
launcher.sh
ls_extract.sh
```

Remarque : extraire des fichiers en excluant de certains dossiers ".idea venv .node* **pycache**"

```
tar --exclude='venv*' -czvf  ${file}
tar --exclude='venv*' --exclude='.idea*' --exclude='__pycache__*'   --exclude='*.node*'   -xzf  ${file}

unzip  ${file} -x ".idea*"  "*venv*" "*.node*"  "*__pycache__*"
```

5. Pour la suite, si une commande pour extraire l'archive est affichée, créer un dossier au nom de l'archive

6. Ajout d'un fichier dans chaque dossier

on désire ajouter dans chaque dossier créé par la commande précédente (désarchiver dans la réalité) le fichier **`launcher.sh`**

exemple de lanceur **`launcher.sh`**

```
sed -i 's/host=.*/host="serveurmysql",/g' app.py
sed -i 's/user=.*/user="votreLogin",/g' app.py
sed -i 's/password=.*/password="motDePasse",/g' app.py
sed -i 's/database=.*/database="BDD_sae",/g' app.py

mysql --user=votreLogin --password=motDePasse --host=serveurmysql BDD_sae < sql_projet.sql

python app.py
```

Prendre un projet flask, un fichier sql et tester le lanceur.

5 Exercice 8

on dispose d'un fichier csv **`login.csv`**

```
aduboit;mdp1
bgrange;mdp2
cdurand;mdp3
dgregoire;mdp4
eroi;mdp5
```

Ecrire un script bash **`creer_compte_csv.sh`**

ce script doit :

- vérifier que le compte n'existe pas encore dans le fichier **`/etc/password`**
- si le compte existe déjà
  - indiquer dans la sortie standard que le compte existe déjà
- si le compte n'existe pas, créer le compte utilisateur

Conseil : rechercher des informations sur la commande **`cut`** puis tester la commande **`NAME=$(echo $LIGNE | cut -d; -f1)`**

- https://lipn.univ-paris13.fr/~cerin/SE/S2SE_01_LectureFichiersShell2.html
- https://www.formatux.fr/formatux-bash/module-020-mecanismes-base/index.html