Programmation Shell – Manuel d'exercices

Exercice 1 : Redirection vers des fichiers

Exécutez à la suite des commandes qui enregistrent dans un fichier stockage.txt les informations issues des commandes mount, lsblk et df -h. Examinez le fichier produit.

Note : la commande getent permet d'obtenir (entre autres) des informations sur les comptes et groupes du système.

Utilisez cat pour envoyer dans un fichier output.txt les contenus des fichiers /etc/timezone, /etc/shells. Ensuite ajoutez à ce fichiers le contenu des sorties des commandes getent passwd et getent group. Examinez le fichier produit.

Exercice 2 : Joker dans les noms de fichiers

Créez avec touch des fichiers nommés data01, data02, ..., data12. Utilisez mv pour déplacer data01 jusqu'à data07 uniquement dans Documents. Vérifiez.

Utilisez rm pour supprimer tout fichier de la forme data1x où x pourrait valoir de 0 à 9. Vérifiez.

Quelles commandes dans /usr/bin commencent par un z suivi de trois caractères quelconques et se terminent par p ?

Que se passe-t-il si aucun nom ne correspond à une expression ? Réessayez après avoir exécuté shopt -s failglob. Pensez-vous que ce comportement est préférable ?

Exercice 3 : grep et expressions régulières

Utilisez ls et grep ensembles pour lister tous les noms de liens symboliques, puis de répertoires, puis de fichiers « normaux » du répertoire courant

Ajoutez les définitions d'alias correspondant dans votre .bashrc sous les noms lsymb, ldir et lnorm.

Examinez le contenu du fichier mydata. Utilisez grep pour extraire les lignes :

- Contenant un chiffre
- Contenant deux chiffres
- Contenant un nombre entier
- Évoquant un chien, ou un chat, ou les deux
- Les lignes contenant data0NN
- Les lignes contenant data0NN où NN va de 02 à 07
- Les lignes contenant une adresse e-mail valide (attention à faire ceci soigneusement en production, n'hésitez pas à rechercher une expression régulière validée !)

Parcourez les pages de manuel des commande du, sort et head.

Comment du peut-il rapporter la taille occupée sur disque par un ou plusieurs répertoire ? Comment sort peut-il trier son entrée standard selon des critère numérique descendant ?

En une seule ligne de commande affichez les tailles et chemins vers les trois sous-répertoires les plus volumineux de /usr.

Exercice 4 : tests simples

Exécutez une ligne de commande qui affiche ok si un fichier monprog est présent dans /tmp et est exécutable. Testez.

Placez cette ligne dans un script test_monprog, rendez-le exécutable et testez-le.

Rappel : interaction utilisateur

Pour stocker une information saisie par l'utilisateur vous pouvez utiliser read -p message variable. La valeur est ensuite accessible via $variable.

test permet aussi de comparer deux variables, comme texte (chaînes de caractères) ou par des comparaisons numériques. Écrivez un script qui demande un identifiant (login) d'utilisateur et affiche "C'est moi" si c'est... vous, ensuite il demande un âge et s'il est supérieur à 42 affiche "Ok boomer!".

Exercice 5 : Boucle for

Écrivez un script qui extrait de la sortie de getent passwd les informations sur les utilisateurs actuellement connectées.

Écrivez un script qui accepte une suite d'identifiants d'utilisateurs (login) et affiche « xxx connecté » pour ceux qui sont connectés.

Écrivez un script qui accept une liste de noms de machines ou d'adresses IP et lance un ping -c 1 ... vers chacun de ces hôtes.

Consultez l'annexe du manuel d'exercice pour trouver de quelle(s) façon(s) for peut être utilisé pour parcourir une séquence de nombres entiers. Écrivez un script qui affiche un triangle rectange de hauteur passé en paramètre, par exemple :

```
$ ./triangle 5
*
**
***
****
*****
```

Exercice 6 : Boucle while

Écrire un script qui attend qu'un processus nommé xeyes apparaisse et alors le tue (note : utiliser pgrep et kill). Testez-le.

Écrire un script qui affiche sur une même ligne toutes les secondes un caractère « . » indéfiniment. La structure de boucle while et la commande true (qui renvoie toujours un code de succès) sera utile ici, ainsi que la commande sleep.

Vérifiez que vous pouvez interrompre le script avec la séquence de touches CTRL+C.

Interceptez le signal INT en lui associant une commande vide. Comment pouvez-vous « tuer » votre script ?

Exercice 7 : Substitution d'ensembles

Créez avec une ligne de commande simple et la commande touch des fichiers vides nommés de data0 à data9.

Créez de la même façon des fichiers vides nommés de ouputA0, outputA1, ..., outputA9, outputB1, ... jusqu'à outputZ9.

Exercice 8 : Fonctions

Reprenez l'example d'utilisation de getopt, modifiez le pour que l'affichage de la syntaxe du script soit réalisé par une fonction usage().

Définissez dans un script nommé calculatrice des fonctions renvoyant respectivement somme, différence, produit et quotient de leur paramètres. En utilisant la construction case/esac écrivez un programme qui demande la saisie de deux nombres et le choix d'une opération puis affiche le résultat attendu.

Exercice 9 : Bouquet final

Nous souhaitons disposer d'un script qui réalise un audit de notre système sur plusieurs points clés qui sera paramétrable finement via la ligne de commande. Ce script produira un rapport bien présenté qui pourra être envoyé par mail périodiquement aux administrateurs.

Autant que possible nous souhaitons que le script applique les méthodes de programmation structurée (fonction, variables locales) et soit lisible pour en faciliter la maintenance.

Ce script nomme audit acceptera les options suivantes :

- -u user1,user2,...
- -m taille_max
- -d
- -o rapport.txt

Qui ont les signications suivantes :

-u : spécifie les comptes utilisateurs à examiner -m : spécifie une taille de fichier en octet -d : spécifie qu'on souhaite un rapport sur l'occupation des disques

Le rapport généré (dans un fichier rapport.txt ou en sortie standard du script) contiendra :

Pour tous les utilisateurs mentionnés après l'option -u, l'occupation disque de leur répertoire de connexion, s'il étaient connectés au moment de la création du rapport et les groupes sont ils sont membres.

Sur l'ensemble du système de fichier racine quels sont les fichiers dont la taille est supérieure à la valeur spécifiée après l'option -m.

Si -d est passé en argument, l'espace total et l'espace disponible des disques.

Parmi les commandes dont votre script aura besoin il y aura très probablement : find, df, du, grep, cut, et quelques autres. N'hésitez pas à consulter le manuel UNIX et à demander des indices à votre formateur.

Quick reference to getting started with Bash scripting

Source : Bash scripting cheatsheet *(https://devhints.io/bash)*

Example

```
#!/usr/bin/env bash

name="John"
echo "Hello $name!"
```

Variables

```
name="John"
echo $name  # see below
echo "$name"
echo "${name}!"
```

Generally quote your variables unless they contain wildcards to expand or command fragments.

```
wildcard="*.txt"
option="iv"
cp -$options $wildcard /tmp
```

String quotes

```
name="John"
echo "Hi $name"  #=> Hi John
echo 'Hi $name'  #=> Hi $name
```

Shell execution

```
echo "I'm in $(pwd)"
echo "I'm in `pwd`"  # obsolescent
# Same
```

See Command substitution

Conditional execution

```
git commit && git push
git commit || echo "Commit failed"
```

Functions

```
get_name() {
  echo "John"
}

echo "You are $(get_name)"
```

See: Functions

Conditionals

```
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
fi
```

See: Conditionals

Strict mode

```
set -euo pipefail
IFS=$'\n\t'
```

See: Unofficial bash strict mode

Brace expansion

```
echo {A,B}.js
```

Expression	Description
`{A,B}`	Same as `A B`
`{A,B}.js`	Same as `A.js B.js`

Expression	Description
`{1..5}`	Same as `1 2 3 4 5`

See: Brace expansion

Parameter expansions

Basics

```
name="John"
echo "${name}"
echo "${name/J/j}"    #=> "john" (substitution)
echo "${name:0:2}"    #=> "Jo" (slicing)
echo "${name::2}"     #=> "Jo" (slicing)
echo "${name::-1}"    #=> "Joh" (slicing)
echo "${name:(-1)}"   #=> "n" (slicing from right)
echo "${name:(-2):1}" #=> "h" (slicing from right)
echo "${food:-Cake}"  #=> $food or "Cake"


length=2
echo "${name:0:length}"  #=> "Jo"
```

See: Parameter expansion

```
str="/path/to/foo.cpp"
echo "${str%.cpp}"    # /path/to/foo
echo "${str%.cpp}.o"  # /path/to/foo.o
echo "${str%/*}"      # /path/to

echo "${str##*.}"     # cpp (extension)
echo "${str##*/}"     # foo.cpp (basepath)

echo "${str#*/}"      # path/to/foo.cpp
echo "${str##*/}"     # foo.cpp

echo "${str/foo/bar}" # /path/to/bar.cpp


str="Hello world"
echo "${str:6:5}"   # "world"
echo "${str: -5:5}"  # "world"


src="/path/to/foo.cpp"
```

```
base=${src##*/}   #=> "foo.cpp" (basepath)
dir=${src%$base}  #=> "/path/to/" (dirpath)
```

Substitution

Code	Description
`${foo%suffix}`	Remove suffix
`${foo#prefix}`	Remove prefix
`${foo%%suffix}`	Remove long suffix
`${foo##prefix}`	Remove long prefix
`${foo/from/to}`	Replace first match
`${foo//from/to}`	Replace all
`${foo/%from/to}`	Replace suffix
`${foo/#from/to}`	Replace prefix

Comments

```
# Single line comment
```

```
: '
This is a
multi line
comment
'
```

Substrings

Expression	Description
`${foo:0:3}`	Substring *(position, length)*
`${foo:(-3):3}`	Substring from the right

Length

Expression	Description
`${#foo}`	Length of `$foo`

Manipulation

```
str="HELLO WORLD!"
echo "${str,}"   #=> "hELLO WORLD!" (lowercase 1st letter)
echo "${str,,}"  #=> "hello world!" (all lowercase)

str="hello world!"
echo "${str^}"   #=> "Hello world!" (uppercase 1st letter)
echo "${str^^}"  #=> "HELLO WORLD!" (all uppercase)
```

Default values

Expression	Description
`${foo:-val}`	`$foo` , or `val` if unset (or null)
`${foo:=val}`	Set `$foo` to `val` if unset (or null)
`${foo:+val}`	`val` if `$foo` is set (and not null)
`${foo:?message}`	Show error message and exit if `$foo` is unset (or null)

Omitting the `:` removes the (non)nullity checks, e.g. `${foo-val}` expands to `val` if unset otherwise `$foo` .

Loops

Basic for loop

```
for i in /etc/rc.*; do
  echo "$i"
done
```

C-like for loop

```
for ((i = 0 ; i < 100 ; i++)); do
  echo "$i"
```

```
done
```

Ranges

```
for i in {1..5}; do
    echo "Welcome $i"
done
```

With step size

```
for i in {5..50..5}; do
    echo "Welcome $i"
done
```

Reading lines

```
while read -r line; do
  echo "$line"
done <file.txt
```

Forever

```
while true; do
  ...
done
```

Functions

Defining functions

```
myfunc() {
    echo "hello $1"
}


# Same as above (alternate syntax)
function myfunc() {
    echo "hello $1"
}
```

```
myfunc "John"
```

Returning values

```
myfunc() {
    local myresult='some value'
    echo "$myresult"
}


result=$(myfunc)
```

Raising errors

```
myfunc() {
  return 1
}


if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

Arguments

Expression	Description
`$#`	Number of arguments
`$*`	All positional arguments (as a single word)
`$@`	All positional arguments (as separate strings)
`$1`	First argument
`$_`	Last argument of the previous command

Note: `$@` and `$*` must be quoted in order to perform as described. Otherwise, they do exactly the same thing (arguments as separate strings).

Conditionals

Conditions

Note that `[[` is actually a command/program that returns either `0` (true) or `1` (false). Any program that obeys the same logic (like all base utils, such as `grep(1)` or `ping(1)`) can be used as condition, see examples.

Condition	Description
`[[ -z STRING ]]`	Empty string
`[[ -n STRING ]]`	Not empty string
`[[ STRING == STRING ]]`	Equal
`[[ STRING != STRING ]]`	Not Equal
---	---
`[[ NUM -eq NUM ]]`	Equal
`[[ NUM -ne NUM ]]`	Not equal
`[[ NUM -lt NUM ]]`	Less than
`[[ NUM -le NUM ]]`	Less than or equal
`[[ NUM -gt NUM ]]`	Greater than
`[[ NUM -ge NUM ]]`	Greater than or equal
---	---
`[[ STRING =~ STRING ]]`	Regexp
---	---
`(( NUM < NUM ))`	Numeric conditions

More conditions

Condition	Description		
`[[ -o noclobber ]]`	If OPTIONNAME is enabled		
---	---		
`[[ ! EXPR ]]`	Not		
`[[ X && Y ]]`	And		
`[[ X`		`Y ]]`	Or

File conditions

Condition	Description
`[[ -e FILE ]]`	Exists
`[[ -r FILE ]]`	Readable
`[[ -h FILE ]]`	Symlink
`[[ -d FILE ]]`	Directory
`[[ -w FILE ]]`	Writable
`[[ -s FILE ]]`	Size is > 0 bytes
`[[ -f FILE ]]`	File
`[[ -x FILE ]]`	Executable
---	---
`[[ FILE1 -nt FILE2 ]]`	1 is more recent than 2
`[[ FILE1 -ot FILE2 ]]`	2 is more recent than 1
`[[ FILE1 -ef FILE2 ]]`	Same files

Example

```
# String
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
else
  echo "This never happens"
fi


# Combinations
if [[ X && Y ]]; then
  ...
fi


# Equal
if [[ "$A" == "$B" ]]
```

```
# Regex
if [[ "A" =~ . ]]


if (( $a < $b )); then
   echo "$a is smaller than $b"
fi


if [[ -e "file.txt" ]]; then
  echo "file exists"
fi
```

Arrays

Defining arrays

```
Fruits=('Apple' 'Banana' 'Orange')


Fruits[0]="Apple"
Fruits[1]="Banana"
Fruits[2]="Orange"
```

Working with arrays

```
echo "${Fruits[0]}"           # Element #0
echo "${Fruits[-1]}"          # Last element
echo "${Fruits[@]}"           # All elements, space-separated
echo "${#Fruits[@]}"          # Number of elements
echo "${#Fruits}"             # String length of the 1st element
echo "${#Fruits[3]}"          # String length of the Nth element
echo "${Fruits[@]:3:2}"       # Range (from position 3, length 2)
echo "${!Fruits[@]}"          # Keys of all elements, space-separated
```

Operations

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( "${Fruits[@]/Ap*/}" )          # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
```

```
lines=(`cat "logfile"`)                 # Read from file
```

Iteration

```
for i in "${arrayName[@]}"; do
  echo "$i"
done
```

Dictionaries

Defining

```
declare -A sounds


sounds[dog]="bark"
sounds[cow]="moo"
sounds[bird]="tweet"
sounds[wolf]="howl"
```

Declares `sound` as a Dictionary object (aka associative array).

Working with dictionaries

```
echo "${sounds[dog]}" # Dog's sound
echo "${sounds[@]}"   # All values
echo "${!sounds[@]}"  # All keys
echo "${#sounds[@]}"  # Number of elements
unset sounds[dog]     # Delete dog
```

Iteration

Iterate over values

```
for val in "${sounds[@]}"; do
  echo "$val"
done
```

Iterate over keys

```
for key in "${!sounds[@]}"; do
  echo "$key"
done
```

Options

Options

```
set -o noclobber  # Avoid overlay files (echo "hi" > foo)
set -o errexit    # Used to exit upon error, avoiding cascading errors
set -o pipefail   # Unveils hidden failures
set -o nounset    # Exposes unset variables
```

Glob options

```
shopt -s nullglob    # Non-matching globs are removed  ('*.foo' => '')
shopt -s failglob    # Non-matching globs throw errors
shopt -s nocaseglob  # Case insensitive globs
shopt -s dotglob     # Wildcards match dotfiles ("*.sh" => ".foo.sh")
shopt -s globstar    # Allow ** for recursive matches ('lib/**/*.rb' => 'lib/a/b/c.r
```

Set `GLOBIGNORE` as a colon-separated list of patterns to be removed from glob matches.

History

Commands

Command	Description
`history`	Show history
`shopt -s histverify`	Don't execute expanded result immediately

Expansions

Expression	Description
`!$`	Expand last parameter of most recent command
`!*`	Expand all parameters of most recent command
`!-n`	Expand `n` th most recent command

Expression	Description
`!n`	Expand `n` th command in history
`!<command>`	Expand most recent invocation of command `<command>`

Operations

Code	Description
`!!`	Execute last command again
`!!:s/<FROM>/<TO>/`	Replace first occurrence of `<FROM>` to `<TO>` in most recent command
`!!:gs/<FROM>/<TO>/`	Replace all occurrences of `<FROM>` to `<TO>` in most recent command
`!$:t`	Expand only basename from last parameter of most recent command
`!$:h`	Expand only directory from last parameter of most recent command

`!!` and `!$` can be replaced with any valid expansion.

Slices

Code	Description
`!!:n`	Expand only `n` th token from most recent command (command is `0` ; first argument is `1`)
`!^`	Expand first argument from most recent command
`!$`	Expand last token from most recent command
`!!:n-m`	Expand range of tokens from most recent command
`!!:n-$`	Expand `n` th token to last from most recent command

`!!` can be replaced with any valid expansion i.e. `!cat` , `!-2` , `!42` , etc.

Miscellaneous

Numeric calculations

```
$((a + 200))      # Add 200 to $a

$(($RANDOM%200))  # Random number 0..199

declare -i count  # Declare as type integer
count+=1          # Increment
```

Subshells

```
(cd somedir; echo "I'm now in $PWD")
pwd # still in first directory
```

Redirection

```
python hello.py > output.txt            # stdout to (file)
python hello.py >> output.txt           # stdout to (file), append
python hello.py 2> error.log            # stderr to (file)
python hello.py 2>&1                    # stderr to stdout
python hello.py 2>/dev/null             # stderr to (null)
python hello.py >output.txt 2>&1        # stdout and stderr to (file), equivalent to
python hello.py &>/dev/null             # stdout and stderr to (null)
echo "$0: warning: too many users" >&2  # print diagnostic message to stderr

python hello.py < foo.txt      # feed foo.txt to stdin for python
diff <(ls -r) <(ls)            # Compare two stdout without files
```

Inspecting commands

```
command -V cd
#=> "cd is a function/alias/whatever"
```

Trap errors

```
trap 'echo Error at about $LINENO' ERR
```

or

```
traperr() {
  echo "ERROR: ${BASH_SOURCE[1]} at about ${BASH_LINENO[0]}"
}

set -o errtrace
trap traperr ERR
```

Case/switch

```
case "$1" in
  start | up)
    vagrant up
    ;;

  *)
    echo "Usage: $0 {start|stop|ssh}"
    ;;
esac
```

Source relative

```
source "${0%/*}/../share/foo.sh"
```

printf

```
printf "Hello %s, I'm %s" Sven Olga
#=> "Hello Sven, I'm Olga

printf "1 + 1 = %d" 2
#=> "1 + 1 = 2"

printf "This is how you print a float: %f" 2
#=> "This is how you print a float: 2.000000"

printf '%s\n' '#!/bin/bash' 'echo hello' >file
# format string is applied to each group of arguments
printf '%i+%i=%i\n' 1 2 3  4 5 9
```

Transform strings

Command option	Description
`-c`	Operations apply to characters not in the given set

Command option	Description
`-d`	Delete characters
`-s`	Replaces repeated characters with single occurrence
`-t`	Truncates
`[:upper:]`	All upper case letters
`[:lower:]`	All lower case letters
`[:digit:]`	All digits
`[:space:]`	All whitespace
`[:alpha:]`	All letters
`[:alnum:]`	All letters and digits

Example

```
echo "Welcome To Devhints" | tr '[:lower:]' '[:upper:]'
WELCOME TO DEVHINTS
```

Directory of script

```
dir=${0%/*}
```

Getting options

```
while [[ "$1" =~ ^- && ! "$1" == "--" ]]; do case $1 in
  -V | --version )
    echo "$version"
    exit
    ;;
  -s | --string )
    shift; string=$1
    ;;
  -f | --flag )
    flag=1
    ;;
esac; shift; done
if [[ "$1" == '--' ]]; then shift; fi
```

Heredoc

```
cat <<END
hello world
END
```

Reading input

```
echo -n "Proceed? [y/n]: "
read -r ans
echo "$ans"
```

The `-r` option disables a peculiar legacy behavior with backslashes.

```
read -n 1 ans    # Just one character
```

Special variables

Expression	Description
`$?`	Exit status of last task
`$!`	PID of last background task
`$$`	PID of shell
`$0`	Filename of the shell script
`$_`	Last argument of the previous command
`${PIPESTATUS[n]}`	return value of piped commands (array)

Go to previous directory

```
pwd # /home/user/foo
cd bar/
pwd # /home/user/foo/bar
cd -
pwd # /home/user/foo
```

Check for command's result

```
if ping -c 1 google.com; then
  echo "It appears you have a working internet connection"
fi
```

Grep check

```
if grep -q 'foo' ~/.bash_history; then
  echo "You appear to have typed 'foo' in the past"
fi
```