Fundamentos de Sistemas Operativos / Sistemas de Operação

(Academic year 2024-2025)

Guiões das aulas práticas

Script #06 — The bash scripting language

Summary

Shell scripting using bash.

Exercise 1 *Familiarization with common Unix/Linux commands.*

Commands can be executable programs or builtin `bash` *commands. The former are files with executable permissions. The latter are functions internal to the* `bash` *program.*

1. *Command* `man` *shows a description page for the (file) command passed as argument. Use it to see the role of commands. It follows a list of common ones:* `ls`, `mkdir`, `rmdir`, `pwd`, `rm`, `mv`, `cat`, `echo`, `less`, `head`, `tail`, `find`, `cp`, `diff`, `wc`, `sort`, `grep`, `sed`, `tr`, `cut`, `paste`, `chmod`, `stat`, `history`, *Try to see the manual page of some of them.*

2. *Commands usually have switches/options that allow you to change their default behavior. It follows some examples using command* `ls`.

```
# executed without options,
# ls prints the list of not hidden files in the current directory
touch .dummy    # to create a file named ".dummy"
ls              # file ".dummy" does not show up

# option -a includes all hidden files in the list
ls -a           # now, file ".dummy" appears in the list, but also . and ..

# option -A includes all hidden files except . and ..
ls -A

# option -l prints file properties along with the name
ls -l           # not including the hidden files
ls -la          # including also the hidden files

# option -t prints files sorted by modification time, newest first
ls -lt
```

3. *Command* `help` *shows a description page for the internal* `bash` *command passed as argument.*

```
# the following command shows a list of all bash commands
help

# the following comand shows a description of the internal command cd
help cd
```

Exercise 2 *Redirecting input and output.*

Many commands receive input from the standard input *(usually the keyboard) and deliver output to the* standard output *(usually the terminal display window). Error messages are usually and preferably sent to the* standard error *(also usually the terminal display window). However, it is possible to change this default behaviour.*

1. *Operators* `>` *and* `>>` *redirect the* standard output *to a given file, the former creating a new file, the latter appending to the end of an existing one. The following sequence of commands illustrates their use.*

```
# message "ola" is sent to the terminal display window
echo ola

# message "ola" is sent to file "z",
# any previous content of it being deleted
echo ola > z

# message "ola" is appended to the end of file "z";
# file "z" is created if it does not exit yet
echo ola >> z
```

2. *Operators* `2>` *and* `2>>` *redirects the* standard error *to a given file. The following sequence of commands illustrates its use.*

```
rm -f zzz          # to guarantee file "zzz" does not exist

# an error message is sent to the terminal display window
cat zzz

# the error message is sent to file "z",
# any previous content of it being deleted
cat zzz 2> z
cat z              # to check the error message is there

# the error message is appended to the end of file "z"
cat zzz 2>> z      # the error message is appended to the end of file "z"
cat z              # Why are there 2 error messages in file "z"?
cat zzz > z        # the error message is sent to the terminal window. Why?
```

3. *Operator* `|` *redirects the* standard output *of a command to the* standard input *of the following one, an operation known as the piping of commands. The following sequence of commands illustrates its use.*

```
# the following command sends to the terminal display window
# the constents of file "/etc/passwd"
cat /etc/passwd

# the following piping of commands prints in the terminal display window
# the number of lines of file "/etc/passwd"
cat /etc/passwd | wc -l
```

4. *Operators* `2>&1` *and* `>&2` *(or* `1>&2`*) redirect the* standard error *to the* standard output *and the* standard output *to the* standard error*, respectively. The following sequence of commands illustrates their use.*

```
rm -f zzz                   # to guarantee zzz does not exist

# in the following command, the standard error is not redirected,
# so the error message appears in the terminal
cat zzz > err

# in the following command, the standard error is redirected
# to the same file as the standard output,
# so the error message is sent to file 'err'
cat zzz > err 2>&1
cat err                     # to check the error message is there

# in the following command, the standard output is not redirected
cat /etc/passwd 2> z

# in the following command, the standard output is redirected
# to the same file as the standard error
cat /etc/passwd 2> z >&2
cat z                       # to check the it
```

Exercise 3 *Using special characters.*

1. *Character* `*` *is a string wildcard, representing zero or more characters. Character* `?` *is single-character wildcard, representing one character. Analysing the result of the execution of the following* `ls` *commands, try to understand the meaning of characters* `*` *and* `?`

```
mkdir dir1
cd dir1
touch a a1 a2 a3 a11 b b1 b11 b12 b21  # to create some files
ls
ls a*
ls *1
ls a?
ls b??
ls b?*
ls *
```

2. *Characters* `[` *and* `]` *can be used to define subset of character values. Analysing the result of the execution of the following* `ls` *of commands, try to understand their usage.*

```
touch a a1 a2 a3 a11 b b1 b11 c c11  # to create some files
ls
ls [ac]
ls [a-c]
ls [a-c]?
ls [ab]*
```

3. *Character* `\` *can be used to remove the special meaning of the character following it. The following sequence of commands illustrates its use.*

```
touch a1 a2 a3 a4 a22   # to create some files
echo a*
echo a\*
echo a?
echo a\?
echo a\[
echo a\\
```

4. *The weak quote (*`"`*) and strong quote (*`'`*) characters can be used to remove the special meaning of a sequence of characters. The following sequence of commands illustrates their use.*

```
touch a1 a2 a3 a4 a22   # to create some files
echo a*
echo "a*"
echo 'a*'
```

Exercise 4 *Declaring and using variables.*

1. *Variables are supported by* `bash`. *The assignment of a value to a variable is done using the syntax* ≪`id`≫=≪`value`≫, *where* ≪`id`≫ *represents the name of the variable and* ≪`value`≫ *the value to be assigned. No space can exist immediately before and immediately after the* `=` *assign operator. The value of a variable can be retrieved using the syntax* `${`≪`id`≫`}`, *where* ≪`id`≫ *represents the name of the variable. If no confusion arises, the braces can be omitted. The following sequence of commands illustrates their use.*

```
# assign 'abc' to variable x and '0123456789' to variable xx
x=abc
xx=0123456789

# display the values of variables x and xx
echo ${x}
echo $x
echo ${xx}
echo $xx

# display a string composed of the value of x concatenated with string 'x'
echo ${x}x

# display a string composed of the string 'x' concatenated with the value of variable x
echo x${x}
echo x$x

# Can you antecipate the result of executing the following 3 commands?
touch a1 a2 a3 a4 a22   # to create some files
z=a*                    # which value is assigned to z?
ls $z
echo $z
```

2. *In a previous exercise, the weak (*`"`*) and strong (*`'`*) quote characters seem to be equivalent. They are not.*

```
touch a1 a2 a3 a4 a22   # to create some files
z=a*
echo $z
echo "$z"       # reference to variables are expanded within weak quotes
echo '$z'       # reference to variables are not expanded within strong quotes
```

3. *In* `bash`, *there are several ways to manipulate the values of variables The following sequence of commands illustrates two of them*

```
x=0123456789        # to create a variable

# The following construction allows to extract a substring from the value of a variable
echo ${x:2:4}

# The following construction allows to replace a substring with a new value
echo ${x/123/ccc}
```

4. *Use the manpage of* `bash` (`man bash`) *to see other possible manipulations of variables.*

Exercise 5 *Defining and using functions.*

1. *Functions are supported in* `bash`*. You can define them directly from the terminal. The following sequence of commands illustrates the definition and use of a function.*

```
# the following code defines function x
myls()
{
    ls -ltrh
}

# the following code calls the previously defined function myls
myls

# You can use it as any other command.
# Here, piping its output to the word count command
myls | wc -l
```

2. *Functions can accept arguments. Variables* `$1`*,* `$2`*, ...,* `$*`*,* `$@` *and* `$#` *can be used to access them.*

```
# definition of function y
y()
{
    echo $#      # the number of arguments
    echo $1      # the first argument
    echo $2      # the second argument
    echo $*      # the list of all arguments
    echo $@      # idem
    echo "$*"    # idem
    echo "$@"    # idem
}

# calling it with some arguments
y a bb ccc dddd eeeee
y a "b  b" ccc 'dd    dd' eeeee     # note the effect of the quote characters
```

Exercise 6 *Grouping commands.*

1. *Characters* `{ e }` *can be used to group commands.*

```
# In the following example, the output of a group of commands is redirected to a file.
{
    ls
    echo ========================
    ls
} > z
cat z
```

2. *Characters* `( e )` *can also be used to group commands.*

```
# In the following example, the output of a group of commands is redirected to a file.
(
    ls
    echo ========================
    ls
) > z
cat z
```

3. *The difference berween them is that in the second case the execution happens in a new instance of the bash. The following bash code shows the differences. Pay attention to the successive values of variable* `zzz`.

```
# variable zzz within the {..} group is the same as out of it,
# because every thing happens in the same shell scope
zzz=abc
echo "$zzz (out of group)"
{
    echo "$zzz (within group)"
    zzz=xpto
    echo "$zzz (within group)"
}
echo "$zzz (out of group)"

# variable zzz within the (..) group is not the same as out of it,
# so the assignment done within the subshell does not affect variable zzz out of it
zzz=abc
echo "$zzz (out of group)"
(
    echo "$zzz (within group)"
    zzz=xpto
    echo "$zzz (within group)"
)
echo "$zzz (out of group)"
```

Exercise 7 *The conditional* `if` *construction.*

1. *Commands have a return value, that, in C/C++ programs, corresponds to the argument of the* `return` *instruction of the* `main` *function or to the argument of the* `exit` *function. The* `bash` *saves this return value in variable* `$?`*. Execute the following sequence of commands to see it.*

```
ls
echo $?         # should display 0, because the previous command succeed

rm zzz          # to guarantee file zzz does not exist
echo $?         # should display 1, if file zzz does not exist

test -f zzz
echo $?         # should display 1, because file zzz does not exist

touch zzz       # to guarantee file zzz exists
test -f zzz
echo $?         # should display 0, because file zzz exists
```

2. *The value of* `$?` *is used by* `bash` *as a boolean value,* `0` *representing* `true` *and other values representing* `false`*. The following sequence of commands illustrates a use of the* `if .. then .. [else ..] fi` *construction.*

```
touch zzz       # to guarantee file zzz exists
if test -f zzz
then
    echo "File zzz exists"
else
    echo "File zzz does not exist"
fi

check()
{
    if test -f $1
    then
        echo -e "\e[33mFile zzz exists\e[0m"
    else
        echo -e "\e[31mFile zzz does not exist\e[0m"
    fi
}

touch zzz       # to guarantee file zzz exists
check zzz
rm -f zzz       # to guarantee file zzz does not exist
check zzz
```

3. *Function* `test` *can be called using brackets.*

```
check()
{
    if [ -f $1 ]    # the brackets should appear isolated
    then
        echo -e "\e[33mFile zzz exists\e[0m"
    else
        echo -e "\e[31mFile zzz does not exist\e[0m"
    fi
}
```

```
touch zzz     # to guarantee file zzz exists
check zzz
rm -f zzz     # to guarantee file zzz does not exist
check zzz
```

4. *Operator* `!` *is the logical not.*

```
rm -f zzz       # to guarantee file zzz does not exist
if ! test -f zzz
then
    echo "File zzz does not exist"
fi

# or, equivalently
if ! [ -f zzz ]
then
    echo "File zzz does not exist"
fi
```

5. *Operators* `&&` *and* `||` *are simplified conditional constructions.*

```
# the command following \verb!&&! only executes if the previous succeed
touch zzz       # to guarantee file zzz exists
test -f zzz && echo "File zzz exists"

# the command following \verb!||! only executes if the previous fails
rm -f zzz       # to guarantee file zzz does not exist
test -f zzz || echo "File zzz does not exist"
```

Exercise 8 *The multiple choice* `case` *construction.*

1. *The* `case` *construction allows branching based on patterns, as is illustrated by the following code.*

```
z()
{
    case $# in
        0)
            echo "No arguments were given"
            # a double ;; is used to end a branch
            ;;
        1)
            echo "One argument was given"
            ;;
        2|3)
            # a !|! in a pattern defines an alternative
            echo "Two or three arguments were given"
            ;;
        *)
            # a !*! means any value
            # being the last branch, it represents the otherwise values
            echo "More than three arguments were given"
            ;;
    esac
}

z
z aa
z aa bb
z aa bb cc
z aa bb cc dd
z aa bb cc dd ee
```

Exercise 9 *The repetitive* `for` *construction.*

1. *The* `for` *construction allows for iteration through a list of values. Next code appends a prefix to the name of all files in the current directory whose names start with an* `a`*.*

```
touch a1 a2 a77 abc b1 c12 ddd      # to create some files
ls
prefix="_a_"
for f in a1 a2 a77 abc b1 c12 ddd
do
    echo "changing the name of \"$f\""
    mv $f $prefix$f
done
ls
```

2. *The following code creates and uses a function that iterates through all files passed as arguments.*

```
f1()
{
    for file in $*
    do
        echo "==== $file ====" > $file
    done
}

f1 abc xpto zzz
cat xpto
cat abc
cat zzz
```

Exercise 10 *The repetitive* `while` *and* `until` *constructions.*

1. *In the following code, function* `f2` *is equivalent to function* `f1` *of the previous exercise.*

```
f2()
{
    while [ $# -gt 0 ]
    do
        echo "==== $1 ====" > $1
        shift
    done
}

rm -f abc xpto zzz      # to guarantee they do not exist
f2 abc xpto zzz
cat xpto
cat abc
cat zzz
```

2. *In the following code, function* `f3` *is equivalent to function* `f1` *of the previous exercise.*

```
f3()
{
    until [ $# -eq 0 ]
    do
        echo "==== $1 ====" > $1
        shift
    done
}

rm -f abc xpto zzz      # to guarantee they do not exist
f3 abc xpto zzz
cat xpto
cat abc
cat zzz
```

Exercise 11 *Script files.*

1. *You can create a file whose contents is a program in* `bash` *(or a program in any other shell scripting language). To execute it, you can pass its name as an argument to the* `bash` *command or you can change its permissions to include execution. Such program are usually called* shell scripts. *Use your favorite text editor to create file* `myscript.bash`*, with the following contents.*

```
#!/bin/bash
# The previous line (comment) tells the operating system that
#   this script is to be executed by bash
#
# This script selects and sorts the lines of a given file,
#   except the first 5 and the last 5.
#
if [ $# -ne 1 ]
then
    echo "A single argument is mandatory" 1>&2
    exit 1
fi

if ! [ -f $1 ]
then
    echo "Given argument ($1) is not a regular file" 1>&2
    exit 1
fi

head -n -5 $1 | tail -n +6 | sort
```

2. *In the following code, the previous script is called through the* `bash`.

```
bash myscript.bash
bash myscript.bash xpto abc zzz
rm -f abc                # to guarantee file 'abc' do not exist
bash myscript.bash abc

# create a file for testing
seq -w 100 -1 1 > xpto
cat xpto
bash myscript.bash xpto
```

3. *In the following code, permissions are changed to include execution and thus the script can be called directly. Actually, the code is executed within a new* `bash`*, implicitly called.*

```
chmod +x myscript.bash
./myscript.bash xpto
```

Exercise 12 `Bash` *supports both indexed and associative arrays.*

1. *The indices of an indexed array do not need to be contiguous and can not be negative. The following code shows the use of an indexed array.*

```
a[1]=aaa
echo ${a[1]}         # the ${..} is mandatory to access array elements

# the 'declare -a' may be used to said the variable is an array element
declare -a a[2]=bbb

# integer arithmetic expression are allowed
a[2+3]=eee

# there is a construction to represent all elements of an array
echo ${a[*]}

# there is a construction to represent the number of elements of an array
echo ${#a[*]}

# there is a construction to represent the list of element indices of an array
echo ${!a[*]}

# iterating through the list of elements
for v in ${a[*]}
do
    echo $v
done

# iterating through the list of indices
for i in ${!a[*]}
do
    echo "a[$i] = ${a[$i]}"
done
```

2. *Associative arrays need to be declared explicitly. The following code illustrates its declaration and use.*

```
# declaring and populating an associative array (map)
declare -A arr
arr["homem"]=man
arr["papel"]=paper
arr["olá"]=hello
arr["lição"]=lesson

# the same construction as before can be used to manipulate the array
echo ${arr[*]}       # the list of elements in the array
echo ${#arr[*]}      # the number of elements in the array
echo ${!arr[*]}      # the list of indices used in the array

# iterating through the list of indices
for i in ${!arr[*]}
do
    echo "The mapping of \"$i\" is \"${arr[$i]}\""
done
```