Bash Lecture 4 – Bash Scripting beyond the basics

Bibliography and learning materials

★ Bibliography:

https://www.rigacci.org/docs/biblio/online/sysadmin/toc.htm

https://www.tldp.org/LDP/abs/html/

★ Learning Materials:

http://www.ee.surrey.ac.uk/Teaching/Unix/

https://github.com/bertocco/abilita_info_units_2021

Arguments of this lesson

★Redirection

★Bash scripting programming:

- Main programming elements (if, for, while,…)
- Functions
- Scope of variables
- Examples (using files)
- Basic `sed`
- Basic `awk`

Redirection (2)

Redirection [> &> >& >>].

- Redirect stdout to file (overwrite filename if it already exists):

scriptname > filename

scriptname >> filename # appends the output of scriptname to file filename. If
filename does not already exist, it is created

- Redirect stderr to file (overwrite filename if it already exists):

scriptname 2> filename

- Redirect both the stdout and the stderr of command to filename:

command &> filename redirects both the stdout and the stderr of command to filename

- Redirects stdout of command to stderr:

command >&2

- Redirects stderr of command to stdout:

command 2>&1

Redirection: Examples

- Stdout redirected to file

```
find . -name pippo > find-output.txt
```

- Stderr redirected to file

```
find . -name pippo 2> find-errors.txt
```

- discards any errors that are generated by the find command

```
find / -name "*" -print 2> /dev/null
```

/dev/null is a simple device (implemented in software and not corresponding to any hardware device on the system).

/dev/null looks empty when you read from it.

Writing to /dev/null does nothing: data written to this device simply "disappear."

Often a command's standard output is silenced by redirecting it to /dev/null, and this is perhaps the null device's commonest use in shell scripting:

```
command > /dev/null
```

- Redirect both stdout and stderr to file

```
find .  -name pippo &> out_and_err.txt
```

- Redirect stderr to stdout: `find .  -name filename 2>&1`
- Redirect stdout to stderr: `find .  -name filename 1>&2`

Special characters: Pipe

Pipe [|]. Passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell. This is a method of chaining commands together.

```
echo ls -l | sh
#  Passes the output of "echo ls -l" to the shell,
#+ with the same result as a simple "ls -l".
```

```
cat *.lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

A pipe sends the stdout of one process to the stdin of another. In a typical case, a command, such as cat or echo, pipes a stream of data to a command that transforms it in input for processing:

```
cat $filename1 $filename2 | grep $search_word
```

Redirection with pipe and tee examples

Examples of redirection of the output of a command to be used as input of another:

- Display the output of a command (in this case ls) by pages:
  ls -la | less
- Count files in a directory:
  ls -l | wc -l
- Count the number of rows containing of the word “canadesi” in the file vialactea.txt
  grep canadesi vialactea.txt | wc -l
- Count the number of words in the rows containing the word “canadesi”

`tee` is useful to redirect output both to stdout and to a file. Example:
find . -name filename.ext 2>&1 | tee -a log.txt
This will take stdout and append it to log file. The stderr will then get converted to stdout which is piped to tee which appends it to the log and sends it to stdout which will either appear on the tty or can be piped to another command.

To go deep: https://stackoverflow.com/questions/2871233/write-stdout-stderr-to-a-logfile-also-write-stderr-to-screen

Exercise: redirection

Create a directory and file tree like this one:

my_examples /ex1.dir
/ex2.txt
/ex3.dir
/*ex3.dir/*file1.txt
/*ex3.dir/*file2.txt
/*ex3.dir/*file3.txt

Remove read permissions to directory /*ex2.dir*
Redirect output on a file. Error is displayed on terminal
Redirect error on a file. Output is displayed on terminal
Verify the content of the files
Stderr redirected to file
Redirect output and errors symultaneously

Use pipe to redirect the output of a command to another command and to a file
Use tee to redirect output both to stdout and to a file

Redirection (1)

Each UNIX command (or program) is connected to three communication channels between the command and its environment:

- Standard input (stdin) where the command read its input
- Standard output (stdout) where the command writes its output
- Standard error (stderr) where the command writes its error

When a command is executed via an interactive shell, the streams are typically connected to the text terminal on which the shell is running, but can be changed with redirection or with a pipeline

redirect stdout to a file	redirect stderr and stdout to a file
redirect stderr to a file	redirect stderr and stdout to stdout
redirect stdout to stderr	redirect stderr and stdout to stderr
redirect stderr to stdout	

Standard Input, Standard Output and Standard Error Symbols:

standard input	0<
standard output	1>
standard error	2>

Conditional execution

Conditional statements:

- ★If … then
- ★ If … then … else
- ★ If … then … elif
- ★ case

Conditional statement “if...then”

The if construction allows you to specify different courses of action to be taken in a shell script, depending on the success or failure of a command.

The most compact syntax of the if command is:

```
if TEST-COMMANDS; then COMMANDS; fi
```

Which is the same, less compact:

```
if TEST-COMMANDS
    then COMMANDS
fi
```

The TEST-COMMAND list is executed, and if its return status is zero (True), the COMMANDS are executed. The return status is the exit status of the last command executed, or zero if the condition tested is not True (different from 0).

Example of conditional statement “if...then”

- Testing exit status

The ? variable holds the exit status of the previously executed command (the most recently completed foreground process).

Example

Test to check if a command has been successfully executed:

```
ls -l
if [ $? -eq 0 ]
   then echo 'That was a good job!'
fi
```

- Numeric comparisons

The example below use numerical comparisons:

```
num=`less work.txt |wc -l`
echo $num
If [[  "$num" -gt "150"  ]]
then echo ; echo "you've worked hard enough for today."
fi
```

Main conditional operators

Relational operators

- -lt (<) lower-than
- -gt (>) greather-then
- -le (<=) lower-equal
- -ge (>=) greather-equal
- -eq (==) equal
- -ne (!=) not equal

Boolean operators

- && and
- || or
- | not

Files operators:

- if [-x "$filename"]; then # if filename is executable
- if [-e "$filename"]; then # if filename exists
- ……

Condition check

The [[ ]] construct is the more versatile Bash version of [ ].
This is the extended test command.

No filename expansion or word splitting takes place between [[and]],
but there is parameter expansion and command substitution.

```
file=/etc/passwd
if [[ -e $file ]]
then
  echo "Password file exists."
fi
```

Using the [[...]] test construct, rather than [...] can prevent many logic errors in scripts. For example, the &&, ||, <, and > operators work within a [[ ]] test, despite giving an error within a [ ] construct.

Exercise: True and false result

```
a=3

((($a>10)))
echo $?          # print 1 because the condition is false

((($a>2)))
echo $?          # print 0 because the condition is true
```

Strings comparison example (try)

```
 #!/bin/bash
s1='string'
s2='String'
if [ $s1 == $s2 ]
then
        echo "s1 ('$s1') is not equal to s2 ('$s2')"
fi
If [ $s1 == $s1 ]
then
        echo "s1('$s1') is equal to s1('$s1')"
fi
```

Be careful: the use of if [$s1 = $s2] can be dengerous:
if one of the two strings is empty, a syntax error will be thrown.
Use instead:
 X$1 == x$2 or "$1" == "$2"

Check if a variable is empty example

Try:

```
if [[ X == X$variable_to_check ]]
  then
      echo “variable is empty”
 else
      echo “variable value is $variable_to_check”
fi
```

Then try:

```
variable_to_check=”I_am_not_empty”
if [[ X == X$variable_to_check ]]
  then
      echo “variable is empty”
 else
      echo “variable value is $variable_to_check”
fi
```

Nested conditional if...then statement

```
a=3

if [ "$a" -gt 0 ]
then
  if [ "$a" -lt 5 ]
  then
    echo "The value of \"a\" lies somewhere between 0 and 5."
  fi
fi

# Same result as:

if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]
then
  echo "The value of \"a\" lies somewhere between 0 and 5."
fi
```

Conditional statement “if...then...else”

```
if [ condition-true ]
then
   command 1
   command 2
   ...
else  # Adds default code block executing if original condition tests false.
   command 3
   command 4
   ...
fi
```

Note:
When if and then are on same line in a condition test, a semicolon must terminate the if statement. Both if and then are keywords. Keywords (or commands) begin statements, and before a new statement on the same line begins, the old one must terminate.

Exercise: “if...then...else”

Write a simple example of the construct if...then...else

Suggestion:
Basic example of if .. then ... else:

```
#!/bin/bash
if [ "foo" = "foo" ]; then
   echo expression evaluated as true
else
   echo expression evaluated as false
fi
```

Example of condition with variables:

```
#!/bin/bash
t1="foo"
t2="bar"
if [ "$t1" = "$t2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

Conditional statement “else if and elif”

elif is a contraction for else if. The effect is to nest an inner if/then construct within an outer one.

```
if [ condition1 ]
then
   command1
   command2
else if [ condition2 ]
then
   command3
   command4
else
   default-command
fi
```

```
if [ condition1 ]
then
   command1
   command2
elif [ condition2 ]
then
   command3
   command4
else
   default-command
fi
```

Exercise: “else if and elif”

Translate the previously seen “Nested if...then” example in an “if...elif” form

Case

The BASH CASE statement takes some value once and test it multiple times.
Use the CASE statement if you need the IF-THEN-ELSE statement with many ELIF elements.
Syntax:

```
case $variable in
    pattern-1)
        commands
        ;;
    pattern-2)
        commands
        ;;
    pattern-3|pattern-4|pattern-5)
        commands
        ;;
    pattern-N)
        commands
        ;;
    *)
        commands
        ;;
```

Exercise: case

```
#!/bin/bash
printf 'Which Linux distribution do you know? '
read DISTR

case $DISTR in
    ubuntu)
        echo "I know it! It is an operating system based on Debian."
        ;;
    centos|rhel)
        echo "Hey! It is my favorite Server OS!"
        ;;
    windows)
        echo "Very funny..."
        ;;
    *)
        echo "Hmm, seems i've never used it."
        ;;
esac
```

Loops

Loop statements:

- ★ for
- ★ while
- ★ until

for loop

Executes an iteration on a set of words.
It is slightly different from other languages (like C) where the iteration is done respect to a numerical index.

Syntax:
```
for CONDITION; do
    COMMANDS
done
```

Examples:
```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

C-like for:
```
#!/bin/bash
for i in `seq 1 10`;
do
        echo $i
done
```

for loop examples (try)

Counting:

```
#!/bin/bash
for i in {1..25}
do
   echo $i
done
```

or:

```
#!/bin/bash
for ((i=1;i<=25;i+=1)
do
   echo $i
done
```

Counting on "n" steps

```
#!/bin/bash
for i in {0..25..5}
do
   echo $i
done
```

That will count with 5 to 5 steps.

Counting backwards

```
#!/bin/bash
for i in {25..0..-5}
do
   echo $i
done
```

Acting on files

```
#!/bin/bash
for file in ~/*.txt
do
echo $file
done
```

That example will just list all files with "txt" extension. It is the same as ls *.txt

Calculate prime numbers

```
#!/bin/bash
read -p "How many prime
numbers ?: " num
c=0
k=0
n=2

numero=$[$num-1]
while [ $k -ne $num ]; do
    for i in `seq 1 $n`;do
      r=$[$n%$i]
            if [ $r -eq 0 ]; then
                  c=$[$c+1]
            fi
    done

     if [ $c -eq 2 ]; then
           echo "$i"
           k=$[$k+1]
      fi
      n=$[$n+1]
      c=0
done
```

break statement in for loop

break statement is used to break the loop before it actually finish executing.
You are looking for a condition to be met, you can check the status of a variable for that condition.
Once the contidition is met, you can break the loop. Pseudo-code example:

```
for i in [series]
do
        command 1
        command 2
        command 3
        if (condition) # Condition to break the loop
        then
                command 4 # Command if the loop needs to be broken
                break
        fi
        command 5 # Command to run if the "condition" is never true
done
```

With the use of if ... then you can insert a condition, and when it is true, the loop will be broken with the break statement

continue statement in for loop

continue stop the execution of the commands in the loop and jump to the next value in the series. It is similar to continue which completely stop the loop.

Pseudo-code example:

```
for i in [series]
do
    command 1
    command 2
    if (condition) # Condition to jump over command 3
            continue # skip to the next value in "series"
    fi
    command 3
done
```

break statement in iteration

break command is used to exit out of current loop completely before the actual ending of loop.
Break command can be used in scripts with multiple loops. If we want to exit out of current working loop whether inner or outer loop, we simply use break but if we are in inner loop & want to exit out of outer loop, we use break 2.

Example

```
#!/bin/bash
# Breaking outer loop from inner loop
for (( a = 1; a < 5; a++ ))
do
echo “outer loop: $a”
for (( b = 1; b < 100; b++ ))
do
if [ $b –gt 4 ]
then
break 2
fi
echo “Inner loop: $b ”
done
done
```

The script start with a=1 & move to inner loop and when it reaches b=4, it break the outer loop.

Exercise:
In this same script, use break instead of break 2, to break inner loop & see how it affects the output.

continue statement in iteration

continue command is used in script to skip current iteration of loop & continue to next iteration of the loop.

Example

```
#!/bin/bash
# using continue command
for i in 1 2 3 4 5 6 7 8 9
do
if [ $i –eq 5 ]
then
echo “skipping number 5”
continue
fi
echo “I is equal to $i”
done
```

while loop

Executes one or more instructions while a condition is true.
It stops when the control condition is true or when the execution is intentionally stopped by the programmer with an explicit interruption instruction (break or continue)

Syntax:

```
while CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=0
while [  $counter -lt 10 ]; do
    echo The counter is $counter
    let counter=counter+1
done
```

Example of break statement in while loop

Interrupt the loop at number … (try)

```
#!/bin/bash

num=1
while [ $num -lt 10 ]
do
if [ $num -eq 5 ]
then
echo “$num equal to 5 so I interrupt the loop”
break
fi
echo $num
let num+=1
done
echo “Loop is complete”
```

until loop

Executes one or more instructions until a condition is false.

Syntax:

```
until CONDITION; do
    COMMANDS
done
```

Example:

```
#!/bin/bash
counter=20
until [  $counter -lt 10 ]; do
    echo counter $counter
    let counter-=1
done
```

until vs. while

Until is similar to while, but it is a slightly difference:
Until is executed while the condition is false,
While is executed while the condition is true.
What means it?

Try the following code and check the output:

```
num=1
while [[ $num -lt 10 ]]
do
if [[ $num -eq 5 ]]
then
break
fi
echo $num
let num=num+1
done
echo “Loop while is complete”
```

```
num1=1
until [[ $num1 -lt 10 ]]
do
if [[ $num1 -eq 5 ]]
then
break
fi
echo $num1
let num1=num1+1
done
echo “Loop until is complete”
```

Functions

Functions are use to group sets of commands logically related making them reusable without the need to re-write them.

A function does not need to be declared.

Function example:

```
#!bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
hello
quit
echo foo
```

Syntax:

```
function func_name {
    command1
    command2
    .....
}
```

How to call the function in a script:

```
func_name
```

Functions parameters/arguments

Parameters does not need to be declared.
It is good practice

- to put a comment before the function definition describing parameters and their meaning
- Read the parameters at the beginning of the function

Function with parameters example:

```
#!/bin/bash
function quit {
    exit
}
# input parameter msg=”a message”
function my_func {
    msg=$1
   echo $msg
}
my_func Hello
my_func World
quit
echo foo
```

Syntax with parameters:

```
function func_name {
        command1
        command2
        …..
}
```

How to call the function with parameters in a script:

```
func_name para1 param2 ...
```

Add help to a script

```
cat usage.sh

#!/bin/bash

display_usage() {
    # echo "This script must be run with super-user privileges."
    echo -e "\nUsage:\n$0 [arguments] \n"
}

# if less than two arguments supplied, display usage
if [[  $# -le 1 ]]
  then
    display_usage
    exit 1
fi
```

Add help to a script

Example

```
#!/bin/bash
if [ -z "$1" ]; then        # check if one parameter exists
    echo usage: $0 directory
    exit
fi
srcd=$1
bakd="/tmp/"
mkdir $bakd
of=home-$(date +%Y%m%d).tgz
tar -czf $bakd$of $srcd
```

Positional parameters

Positional parameters are a series of special variables ($0 through $9) that contain the contents of the command line.
If my_script is a bash shell script, we could read each item on the command line because the positional parameters contain the following:
$0 would contain "some_program"
$1 would contain "parameter1"
$2 would contain "parameter2"
…..

This way, if I call my_script with two parameters:
my_script Hello world
Then inside the script I can read them with:

```
#!/bin/bash
script_name=$0
first_word=$1
second_word=$2
Echo “$script_name says $first_word $second_word
```

The mechanism is the same to read functions parameters.

Read the user's input examples

- Example on how to read the user's input:

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

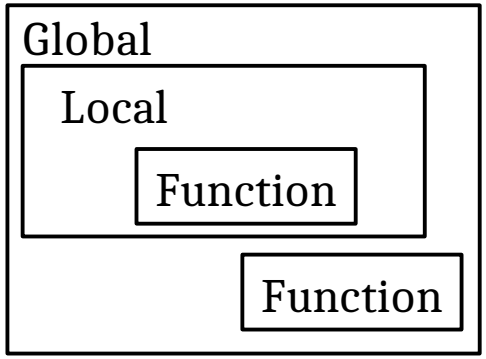
- Example on how to read multiple user's input:

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
echo "How are you?"
```

Scope of variables

In general you can distinguish between

Global
Local
Function
Scope



Bash (like Python) doesn't have block scope in conditionals.

It has local scope within functions, it is also possible to use the ‘local’ modifier which is a keyword to declare the local variables.
Local variables are visible only within the block of code.

Variable scope (visibility) is related mainly to the shell.
Exported variables are visible in all subshells.

Scope of variables

A variable exported is a global variable.

A variable defined in the main body of the script is called a local variable.
- It will be visible throughout the script,
- A variable which is defined inside a function is local to that function.
- It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

- Global variables can have unintended consequences because of their wide-ranging effects: we should almost never use them

Exercise: Scope of variables

```
#!/bin/bash
e=2
echo At beginning e = $e
function test1() {
  e=4
  echo "hello. Now in the function1 e = $e"
}
function test2() {
  local e=4
  echo "hello. Now in the function2 e = $e"
}
test1
echo "After calling the function1 e = $e"

e=2
echo In the file before to call func2 reassign e = $e
test2
echo "After calling the function2 e = $e"
```

Justify the result !

Sed

Sed is a non interactive editor.
It is generally used to parse and transform text, using a simple, compact programming language.

It allows to modify a file usinf scripts with instructions for sed editing plus the filename. Example of string substitution:

```
$sed 's/old_text/new_text/g' /tmp/testfile
```

Sed substitute the string 'old_text' with the string 'new_text' reading from file /tmp/testfile. The result is redirected to stdout, but it can be redirected also to a file using '>'

```
$sed 12, 18d /tmp/testfile
```

Sed displays all the rows from 12 to 18. The original file is not modified by this command, but if you redirect stdout on a new file, if is different from the original one (try).

Awk

Awk match a string on the base of a regular expression and execute a required action:

Create a file /tmp/filetext as follow:

```
cat filetext <
test123
test
Tteesstt
EOF
        $awk '/test/ {print}' /tmp/filetext
test123
test
```

The regular expression requires to match the string 'test'

The required action is to 'print'the string containing ‘test’ when found.

```
        $awk '/test/ {i=i+1} END {print i}' /tmp/filetext
3
```

How to check your scripts

Create a script which launch one of the script you wrote by exercise,
Test the output of the command,
Write if the execution is ok or not.

How to check your scripts

Create a script which launch one of the script you wrote by exercise,
Test the output of the command,
Write if the execution is ok or not.