Bash Lecture 3 – Bash Scripting

Arguments of this lesson

Bash scripting programming:

- What are scripts
- Exec vs source a script
- Bash variables,assignment, hard and soft quoting
- Functions, variable scope, input parameters
- Special characters
- Read input from command line and files
- Script configuration tips

Shell scripting abilities

Many shells have scripting abilities:

Executes sequentially multiple commands written in a script as if they were typed from the keyboard.

Most shells offer additional programming constructs that extend the scripting feature into a programming language.

What is a script

A script is, in the simplest case, a list of system commands stored in a file.

Place commands in a script is useful

- to avoid having to retype them time and again
- to be able to modify and customize the script for a particular application
- to use the script as a program/command

The sha-bang #!

Every script starts with the sha-bang (#!) at the head, followed by the full path name of an interpreter.

Examples:
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl

This tells your system that the file is a set of commands to be fed to the command interpreter indicated by the path.

The command interpreter executes the commands in the script, starting at the top (the line following the sha-bang line), and ignoring comments.

Execute the script

★ The script execution requires the script has “execute” permissions:

chmod +rx scriptname (gives everyone read/execute permission)
chmod u+rx scriptname (gives only the script owner read/execute permission)

★ The script can be executed issuing:

./scriptname

★ The script can be made available as a command:

- moving the script to /usr/local/bin (as root), making it available to all users as a system wide executable. The script could then be invoked by simply typing scriptname [ENTER] from the command-line.
- Including the directory containing the script in the user's $PATH

Summarizing

1_
chmod u+x script.sh
./script.sh
2_
export PATH=.:$PATH
script.sh
3_
gedit .bashrc
echo $PATH
source .bashrc
echo $PATH
script.sh
4_
source /home/bertocco/script.sh

Exec vs source (1)

Both Sourcing and Executing Will Run Commands in the Script.

Example:

- Write the following script

```
$ cat myScript.sh
#!/bin/bash
echo "Hello, I'm a simple script file."
```

- Try to source it

```
source myScript.sh
```

- Meke it executable and exec it

```
chmod +x myScript.sh
./myScript.sh
```

→ The result is the same.

https://www.baeldung.com/linux/sourcing-vs-executing-shell-script

Exec vs source (2)

Both Sourcing and Executing will Run Commands in the Script.

But:

When a script is “sourced” (source script-name), it is executed in the current shell. So, if we’ve declared new variables and functions in the script, after sourcing it, the variables and functions will be valid in the current shell as well.

When a script is executed, it is executed in a new shell, which is a subshell of the current shell. Therefore, all new variables and functions created by the script will only live in the subshell. After the script is done, the subshell process is terminated, too. Thus, the changes are gone.

https://www.baeldung.com/linux/sourcing-vs-executing-shell-script

Exec vs source in practice (1)

Write the script:

```
#!/bin/bash
echo -e 'This is a script file.\nTo test source vs exec'

UNIVERSITY='Universita` di Trieste'
echo "Now, the variable UNIVERSITY=$UNIVERSITY"

say_university() {
    echo "Hi $1, You are at $UNIVERSITY"
}

say_university "Sara"
```

Exec vs source in practice (2)

Source the script and verify that variables and functions defined in the script are defined in the current shell

```
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ source source_vs_exec.sh
This is a script file.
To test source vs exec
Now, the variable UNIVERSITY=Universita` di Trieste
Hi Sara, You are at Universita` di Trieste
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ echo $UNIVERSITY
Universita` di Trieste
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ say_university
Hi , You are at Universita` di Trieste
```

Exec the script and verify that varables and functions defined in the script are not available in the current shell.

```
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ chmod +x source_vs_exec.sh
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ ./source_vs_exec.sh
This is a script file.
To test source vs exec
Now, the variable UNIVERSITY=Universita` di Trieste
Hi Sara, You are at Universita` di Trieste
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ echo $UNIVERSITY

bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ say_university
say_university: command not found
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ ^C
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$
```

Exercise: a first script

★ Write a script that upon invocation

1) Says “Hello!”
2) shows the time and date
3) The script then saves this information to a logfile

★ Make the script executable

★ Execute the script

★ Make the script available as a command

UNIX Variables

★ Variables are how programming and scripting languages represent data. A variable is a label, a name assigned to a location holding data.

★ Standard UNIX variables are split into two categories:

- environment variables:
  if set at login, are valid for the duration of the session
- shell variables:
  apply only to the current instance of the shell and are used to set short-term working conditions;

By convention, environment variables have UPPER CASE and shell variables have lower case names.

★ Environment variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for variables and if found, will use the values stored.

★ Variables can be set: by the system, by you, by the shell, by any program that loads another program.

bash variables

Variable in bash are untyped.

★ Bash variables are character strings: can contain a number, a character, a string of characters.

★ Depending on context (i.e. depending whether the value of a variable contains only digits or not), bash permits arithmetic operations and comparisons on variables.

★There is no need to declare a variable, just assigning a value to its reference will create it.

bash variables: assignment (1)

It must distinguish between the name (right value) of a variable and its value (left value).

If variable1 is the name of a variable,
then $variable1 is a reference to its value, i.e. the data item it contains.

$variable1 is actually a simplified form of ${variable1}. In contexts where the $variable syntax causes an error, the longer form ${variable} may work.

Referencing (retrieving) the variable value is called variable substitution.

=> No space permitted on either side of = sign when initializing variables.

Example:

```
a=375      # Initialize variable
hello=$a   # No space permitted on either side of = sign when initializing variables.
#     ^ ^
# What happens if there is a space? Bash will treat the variable name as a program to
# execute, and the = as its first parameter. TRY
#
echo hello       # hello ## Not a variable reference, just the string "hello" ...
echo $hello      # 375 ##  This *is* a variable reference, i.e. shows the value.
echo ${hello}    # 375 ##  Likewise a variable reference, as above.
```

assignment disambiguation with {}

In the previous slide: “In contexts where the $variable syntax causes an error, the longer form ${variable} may work”. This is called variable disambiguation.

Example:
If the variable $type contains a singular noun and we want to transform it on a plural one adding an ‘s’, we can't simply add an ‘s’ character to $type since that would turn it into a different variable, $types.
Although we could utilize code contortions such as
echo "Found 42 "$type"s"

the best way to solve this problem is to use curly braces:
echo "Found 42 ${type}s",
which allows us to tell bash where the name of a variable starts and ends

Exercise: bash variables

Try:

1) STR=’Hello World!’
   echo $STR

2) Try assignment and echo the variable content:

   a=5324

   a=(1, 3, 4, 6, 5, “otto”) # array

3) Very simple backup script example:
   OF=/tmp/my-backup-$(date +%Y%m%d).tgz
      tar -czf $OF ./subdir_of_where_i_am

bash variables: assignment examples(2)

```
#!/bin/bash
# With command substitution

a=`echo Hello!`   # Assigns result of 'echo' command to 'a' ...
echo $a

a=`ls -l`         # Assigns result of 'ls -l' command to 'a'
echo $a           # Unquoted, however, it removes tabs and newlines.

echo "$a"         # The quoted variable preserves whitespace.
```

Exercise 3: practice with variables assignment

Try different variable assignments and print the variable content to standard output

- Simple assignment
- Command output assignment

Bash variables: quoting

Quoting means just that, bracketing a string in quotes.
This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a wild card character in Regular Expressions).

Partial (or soft) quoting consists in enclosing a referenced value in double quotes (" ... "). This does not interfere with variable substitution. Sometimes referred also as "weak quoting."

Full (or hard) quoting consists in using single quotes ('...').
It causes the variable name to be used literally, and no substitution will take place.

Examples (Try):

```
a=352
echo $a    # 352
echo “$a”  # 352
echo ‘$a’  # $a
```

=> Quoting a variable preserves white spaces.

Exercise 2: variables assignment and quoting

In a bash script:

- Assign a variable
- Print the variable value
- Print a string containing the variable value
- Print a string containing the partial quoted variable
- Print the same string fully quoted
- Assign a variable containing multiple spaces
- Print this new variable
- Print this new variable quoted

- Run the script
- Run the script redirecting the output on a file

Functions

Functions are used to group sets of commands logically related making them reusable without the need to re-write them and making the scripts more readable. A Bash function is a block of reusable code designed to perform a particular operation. Once defined, the function can be called multiple times within a script.

Function example:

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

Syntax:

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

or

```
func_name() {
    command1
    command2
    .....
}
```

How to call the function in a script:

```
func_name
```

Defining Bash Functions (1)

Functions may be declared in two different formats:
1_ function name, followed by parentheses. Preferred and more used.

```
function_name () {
  commands
}
```

Single line version:

```
function_name () { commands; }
```

2_ start with the reserved word ‘function’, followed by the function name.

```
function function_name {
  commands
}
```

Single line version:

```
function function_name { commands; }
```

Defining Bash Functions (2)

- The commands between the curly braces ({}) are called the body of the function. The curly braces must be separated from the body by spaces or newlines.
- Defining a function doesn't execute it. To call a bash function use the function name. Commands between the curly braces are executed whenever the function is called in the shell script.
- The function definition must be placed before any calls to the function.
- When using single line "compacted" functions, a semicolon ; must follow the last command in the function.
- Always try to keep your function names descriptive.

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 ...
```

Passing Arguments to Bash Functions

To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space.
It is a good practice to double-quote the arguments to avoid the misparsing of an argument with spaces in it.

- The passed parameters are $1, $2, $3 … $n, corresponding to the position of the parameter after the function's name.
- The $0 variable is reserved for the function's name.
- The $# variable holds the number of positional parameters/arguments passed to the function.
- The $* and $@ variables hold all positional parameters/arguments passed to the function.

https://linuxize.com/post/bash-functions/

Variables Scope

Global variables are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, all variables by default are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the local keyword and can be used only inside that function. You can have local variables with the same name in different functions.

https://linuxize.com/post/bash-functions/

Variables Scope: example of use

```
#!/bin/bash
var1='A'
var2='B'
my_function () {
  local var1='C'
  var2='D'
  echo "Inside function: var1: $var1, var2: $var2"
}
echo "Before executing function: var1: $var1, var2: $var2"
my_function
echo "After executing function: var1: $var1, var2: $var2"
```

The script starts by defining two global variables var1 and var2. Then there is an function that sets a local variable var1 and modifies the global variable var2.

```
bertocco@speranza:~/work/didattica/AbilitaInformaticheUnits/2023/bash3$ ./scope_of_variables.sh
Before executing function: var1: A, var2: B
Inside function: var1: C, var2: D
After executing function: var1: A, var2: D
```

https://linuxize.com/post/bash-functions/

Variables Scope: example of use

From the output above, we can conclude that:

When a local variable is set inside the function body with the same name as an existing global variable, it will have precedence over the global variable.

Global variables can be changed from within the function.

https://linuxize.com/post/bash-functions/

Return Values

Bash functions don’t allow you to return a value when called.

When a bash function completes, its return value is the status of the last statement executed in the function,

- 0 for success
- non-zero decimal number in the 1 - 255 range for failure.

The return status can be specified by using the return keyword, and it is assigned to the variable $?.

The return statement terminates the function. You can think of it as the function’s exit status .

https://linuxize.com/post/bash-functions/

Examples of function's return values (1)

```
#!/bin/bash

my_function () {
  func_result="some result"
}

my_function
echo $func_result
```

Output will be:

```
some result
```

https://linuxize.com/post/bash-functions/

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.

Special characters (1)

★ Special characters have a meaning beyond its literal meaning

Comments [#]. Lines beginning with a # (with the exception of #!)

```
# This line is a comment.
Comments may also occur following the end of a command.
echo "A comment will follow." # Comment here.
Comments may also follow whitespace at the beginning of a line.
   # Note
```

Command separator [semicolon ;] Permits putting two or more commands on the same line.

```
echo hello; echo world
```

Escape [backslash \] This is a mechanism to express litterally a special charactrer.
For example the \ may be used to escape " and ' echoing a string:

```
echo This is a double quote \”  # This is a double quote ”
```

Special characters (2)

Command substitution [backquotes or backticks `]. The `command` construct makes available the output of command for assignment to a variable.

```
a=`pwd`
echo $a   # display the path of your location
```

Wild card [asterisk *]. The * character serves as a "wild card", it matches every filename in a given directory or every character in a string.

Run job in background [and &]. A command followed by an & will run in the background.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

Within a script, commands and even loops may run in the background.

To bring the script in foreground type `fg` or `CTRL Z fg`

To bring the script in background type `fg` or `CTRL Z bg`

Complete reference:

https://www.tldp.org/LDP/abs/html/special-chars.html

Linux wildcards

There are four main wildcards in Linux:

Asterisk (*) – matches one or more occurrences of any character, including no character.

Question mark (?) – represents or matches a single occurrence of any character.

Square brackets ([ ]) – matches any occurrence of the character(s) enclosed in the square brackets.

Curly brackets ({ }) – matches any occurrence of one of the strings enclosed in the square brackets.

Wildcards examples

[akz]	Exactly one character among a, k or z
[0-9]	Exactly one character among 0 and 9
[!123]	Exactly one character that is not 1 or 2 or 3
[!a-e]	Exactly one character that is not a or b or c or d or e
{fasta,pdb}	Exactly one among the two strings fasta and pdb

Exercise: special characters

- Write a commented command and execute it
- Write two commands on the same row and execute them
- Make the echo of a string containing one or more escaped characters
- Make the echo of a command (like ls or pwd) output
- Use wildcard to list all files starting with ‘a’ in your directory
- Download from MS Teams the script loop.sh from folder General/bash_3/examples, make it executable if needed, execute it in background, recall it in foreground, stop it

`read`

`read` is used in shell scripts to read each field from a file and assign them to shell variables.

A field is a string of bytes that are separated by a space or newline character. If the number of fields read is less than the number of variables specified, the rest of the fields are unassigned.

Flag -r to treat a \(backslash) as part of the input record and not as a control character.

`read` Examples

Example following is a piece of shell script code that reads a file by line:

```
while read -r line
do
    printf 'Line: %s\n' "$line"
done < names_list.txt
```

The file name can be indicate also with full path name.

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?"
```

`read` Examples

Example following is a piece of shell script code that reads first name and last name from namefile and prints them:

- create the file

```
cat <<EOF > names_list.txt
Sara Bertocco
Mario Rossi
John Doe
EOF
```

- Read the file by line and print on standard output

```
while read -r lname fname
do
        echo $lname","$fname
done < names_list.txt
```

Software configuration tips

A software application

- must be indipendent from the location

  *e.g. if I run my application in /home/*myhome/test or in */home/*myhome/bin , its behaviour has to be the same

- must not need code modification to be run

  e.g. if I want run my application two times each one with a different value of a parameter, I do not have to modify manually the code to change the value of the parameter, but the code must be written to acquire the parameter value from command line or from a text configuration file

Software configuration tips: indipendence from location

Set an environment variable:

$ export MY_PATH=”/home/myhome/myProject”

Read the environment variable to acquire the data file path

$ cat /home/myhome/bin/analyser.py

```
#!/usr/bin/python3
import os
data_file = os.getenv("MY_PATH")+"/data/sequence_data.txt"
print("Find my data in")
print(data_file)
```

Software configuration tips: read parameters from command line

Write a bash script to launch your application setting the needed environment and reading input parameters from command line:

```
$ cat my_app_launcher.sh
#!/bin/bash
export MY_PATH="/home/myhome/myProject"
echo "Input the number of parameters you want use:"
read param_num
# invoke the application execution
./home/myhome/bin/analyser.py param_num
```