Introduction to Bash Scripting

https://forge.uclouvain.be/barriat/learning-bash

October 10, 2024

CISM/CÉCI Training Sessions

UCLouvain

Linux command line

A Linux terminal is where you enter Linux commands

It's called the **C**ommand **L**ine **U**ser **I**nterface

CLUI is one of the many strengths of Linux :

- allows to be independent of distros (or UNIX systems like OSX)
- allows to easily work remotely (SSH)
- allows to join together simple (and less simple) commands to do complex things and automate **= scripting**

In Linux, process automation relies heavily on scripting. This involves creating a file containing a series of commands that can be executed together

Linux Shell

A **shell** is a program that takes commands from the keyboard and transmits them to the operating system to perform

The main function is to interpret your commands **= language**

Shells have some built-in commands

A shell also supports programming constructs, allowing complex commands to be built from smaller parts **= scripts**

Scripts can be saved as files to become new commands

> many commands on a typical Linux system are scripts

Bash

The **Bash** shell is one of several shells available for Linux

It is the default command interpreter on most GNU/Linux systems. The name is an acronym for the "**B**ourne-**A**gain **SH**ell"

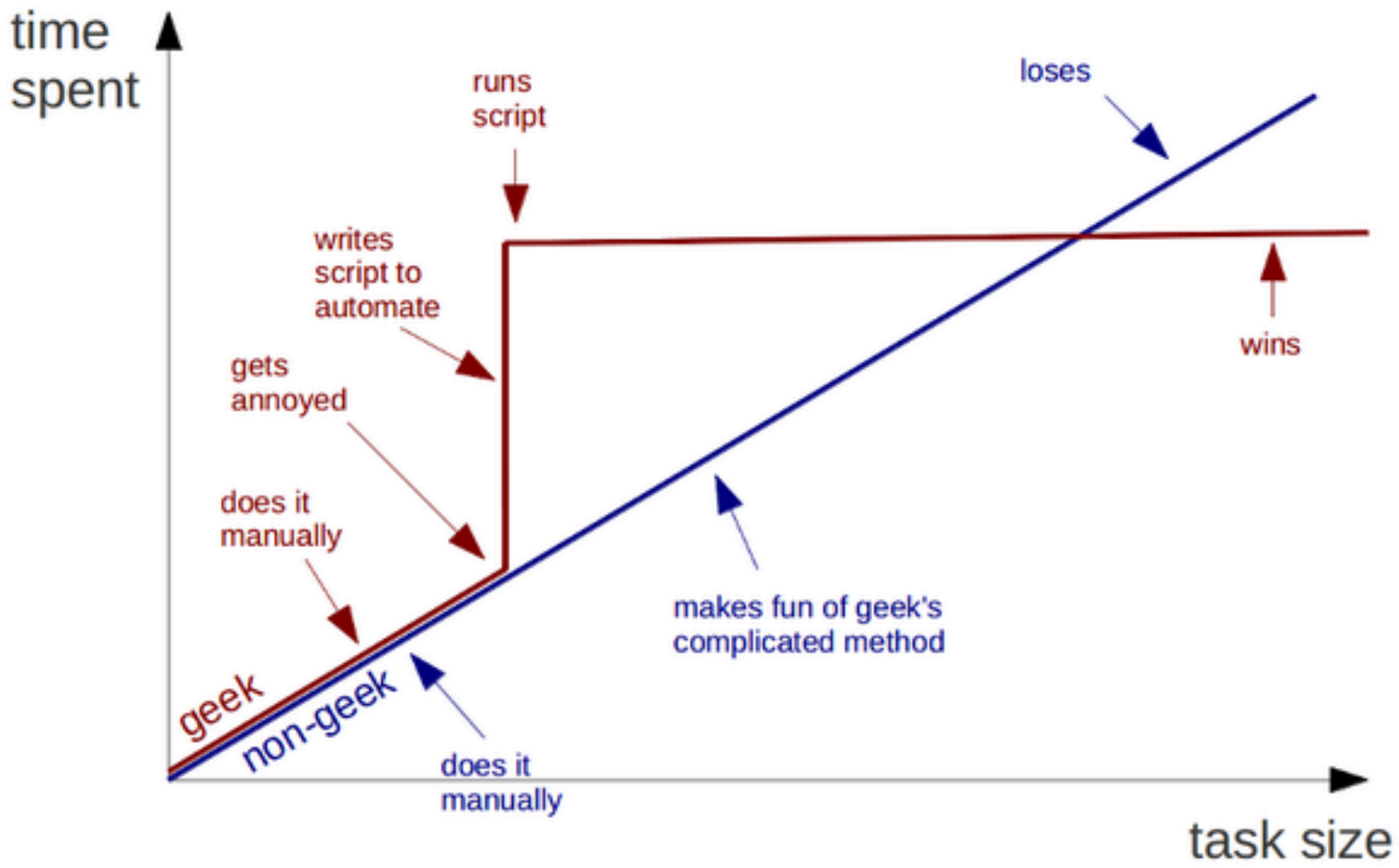
Bash Scripting Demo

```
#!/bin/bash

# declare STRING variable
STRING="Hello World"

# print variable on a screen
echo $STRING
```

Geeks and repetitive tasks



Bash environment

In a Bash shell many things constitute your environment

- the form of your 'prompt' (what comes left of your commands)
- your home directory and your working directory
- the name of your shell
- functions that you have defined
- etc.

Environment includes many variables that may have been set **by bash** or **by you**

Environment variables

Variables	
`USER`	the name of the logged-in user
`HOME`	the user's home directory (similar to `~`)
`PWD`	the current working directory
`SHELL`	the name of the shell

Access the value of a variable by prefixing its name with `$`

So to get the value of `USER` you would use `$USER` in bash code

> You can use special files to control bash variables : `$HOME/.bashrc`

Bash Scripting basics

By naming convention, bash scripts end with `.sh`

> however, bash scripts can run perfectly fine without any extension

A good practice is to define a `shebang` : first line of the script, `shebang` is simply an absolute path to the shell interpreter (see `echo $SHELL` result)

> combination of `bash #` and `bang !`

The usual shebang for bash is `#!/bin/bash`

Comments start with `#`

On a line, any characters after `#` will be ignored (with the exception of `#!`)

```
echo "A comment will follow." # Comment here.
#                             ^ Note whitespace before #
```

There is no standard indentation

- Pick a standard in your team that you can all work to
- Use something your editor makes easy (**Vim** uses `Tab`)

Permissions and execution

- Bash script is nothing else than a **text file** containing instructions to be executed sequentially
  > by default in Linux, a new text file's permissons are **-rw-r--r--** (or 644)
- You can run the script `hello_world.sh` using
  - `sh hello_world.sh`
  - `bash hello_world.sh`
  - `chmod u+x run_all.sh` then `./hello_world.sh`
    > after the `chmod`, you file is **-rwxr--r--** (or 744)

Hands-on exercise

Your first bash script:

1. create a folder `bash_exercises` and go there
2. use your favourite editor (vim, obviously) to create a new file called `exercise_1.sh`
3. write some code in it to display the current working directory as:

   > The current directory is : /home/me/bash_exercises

4. make the file executable
5. run it !

Variables and data types in Bash

Variables let you store data : **numeric values** or **character(s)**

You can use variables to read, access, and manipulate data throughout your script

You don't specify data types in Bash

- assign directly : `greeting="Welcome"` or `a=4`
- assign based on variable: `b=$a`

And then access using `$` : `echo $greeting`

!!! no space before or after `=` in the assignation **!!!**

`myvar = "Hello World"` 💥

Quotes for character(s) `" '`

Double will do **variable substitution**, single will not:

```
$ echo "my home is $HOME"
my home is /home/me
$ echo 'my home is $HOME'
my home is $HOME
```

Command Substitution

```
#!/bin/bash
# Save the output of a command into a variable
myvar=$( ls )
```

Variable naming conventions

- Variable names **should start** with a letter or an underscore
- Variable names can contain letters, numbers, and underscores
- Variable names are **case-sensitive**
- Variable names **should not** contain spaces or **special characters**
- Use descriptive names that reflect the purpose of the variable
- Avoid using **reserved keywords**, such as `if`, `then`, `else`, `fi`, and so on...
- **Never** name your private variables using only **UPPERCASE** characters to avoid conflicts with builtins

String manipulation

Consider `string=abcABC123ABCabc`

- string length : `${#string}` is `15`
- substring extraction :
  - `${string:7}` is `23ABCabc`
  - `${string:7:3}` is `23A`
  - `${string:(-4)}` or `${string: -4}` is `Cabc`

String manipulation

Consider `filename=/var/log/messages.tar.gz`

- substring removal from left :
  - `${filename##/var}` is `/log/messages.tar.gz`
- substring removal from right :
  - `${filename%%.gz}` is `/var/log/messages.tar`

You can use `*` to match all characters:

- `${filename%%.*}` is `/var/log/messages`
- `$(filename##*/)` is `messages.tar.gz`

Arithmetic

Operator	Operation
`+` `-` `\*` `/`	addition, subtraction, multiplication, division
`var++`	increase the variable var by 1
`var--`	decrease the variable var by 1
`%`	modulus (remainder after division)

Several ways to go about arithmetic in Bash scripting :

`let` , `expr` or using **double parentheses**

Arithmetic

```
#!/bin/bash

a=$(( 4 * 5 ))
a=$(( 4 + 5 ))
a=$((3+5))

b=$(( a + 3 ))
echo $b # 11

b=$(( $a + 4 ))
echo $b # 12

(( b++ ))
(( b += 3 ))
echo $b # 16
```

Conditional statements

Use:

- `if condition; then` to start conditional block
- `else` to start alternative block
- `elif` to start alternative condition block
- `fi` to close conditional block

The following operaors can be used beween conditions:

- `||` means **OR**
- `&&` mean **AND**

Conditional exemple

```
#!/bin/bash
num=6

if [ $num -gt 5 ] && [ $num -le 7 ]
then
  echo "$num is 6 or 7"
elif [ $num -lt 0 ] || [ $num -eq 0 ]; then
  echo "$num is negative or zero"
else
  echo "$num is positive (but not 6, 7 or zero)"
fi
```

Operator	Description
`! EXPRESSION`	The EXPRESSION is false
`-n STRING`	The length of STRING is greater than zero
`-z STRING`	The lengh of STRING is zero (ie it is empty)
`STR1 = STR2`	STRING1 is equal to STRING2
`STR1 != STR2`	STRING1 is not equal to STRING2
`INT1 -eq INT2`	INTEGER1 is numerically equal to INTEGER2 (or `==`)
`INT1 -gt INT2`	INTEGER1 is numerically greater than INTEGER2
`INT1 -lt INT2`	INTEGER1 is numerically less than INTEGER2
`INT1 -ne INT2`	INTEGER1 is numerically not equal to INTEGER2

Build conditions with the `test` command

```
test -s /etc/hosts
```

Operator	Description
`-d FILE`	FILE exists and is a directory
`-e FILE`	FILE exists
`-s FILE`	FILE exists and it's size is greater than zero (ie. it is **not empty**)
`-r FILE`	FILE exists and the read permission is granted

> `-w` and `-x` test the write and the execute permission

Conditional: light variation

Check an expression in the `if` statement ?

Use the double brackets just like we did for variables :

```
#!/bin/bash
num=6

if (( $num % 2 == 0 ))
then
  echo "$num is an even number !"
fi
```

Hands-on exercise

1. In your `bash_exercises` folder create a new bash file called `exercise_2.sh` and make it executable
2. Ask the user for two numbers smaller than 100 and put them in variables `NUMBER1` and `NUMBER2`

   ```
   #!/bin/bash
   read NUMBER1
   read NUMBER2
   ```

3. Check if the numbers are smaller than 100
   - If yes, check if both numbers are even and tell the user
   - If not, tell the user (use `echo`)

Arrays

Indexed arrays

```
# Declare an array with 4 elements
my_array=( 'Debian Linux' 'Redhat Linux' Ubuntu OpenSUSE )
# get number of elements in the array
my_array_length=${#my_array[@]}

# Declare an empty array
my_array=( )
my_array[0]=56.45
my_array[1]=568
echo Number of elements: ${#my_array[@]}
# echo array's content
echo ${my_array[2]}
echo ${my_array[@]}
```

Loops

Useful for automating repetitive tasks

Basic loop structures in Bash scripting :

- `while` : perform a set of commands while a test is true
- `until` : perform a set of commands until a test is true
- `for` : perform a set of commands for each item in a list
- controlling loops
  - `break` : exit the currently running loop
  - `continue` : stop this iteration of the loop and begin the next iteration
- last loop mechanism : `select` allows you to create a simple menu system

Examples

```
#!/bin/bash

# Basic while loop
counter=0
while [ $counter -lt 3 ]; do
  echo $counter
  ((counter++))
done
```

```
# range
for i in {1..5}
```

```
# list of strings
words='Hello great world'
for word in $words
```

```
# range with steps for loop
for value in {10..0..2}
```

```
# set of files
for file in $path/*.f90
```

```
# command result
for i in $( cat file.txt )
```

Hands-on exercise

1. In your `bash_exercises` folder create a new bash file called `exercise_3.sh` and make it executable
2. Use the following website to get a list of 10 random words: https://randomwordgenerator.com and put them together in an array
3. Register the start time with `date +%S` and put it in a variable `tstart`
4. Loop over the words and ask the user to give the number of letters. Echo the answers.
5. Register the end time in `tend`
6. Display the total run time and the total number of letters.

Arguments - Positional Parameters

How to pass command-line arguments to a bash script ?

Try a simple example called `test_arg.sh` :

```
#!/bin/bash
echo $1 $2 $4
echo $0
echo $#
echo $@
```

```
bash test_arg.sh a b c d e
```

```
a b d
test_arg.sh
5
a b c d e
```

Special Variables	
`$0`	the name of the script
`$1` - `$9`	the first 9 arguments
`$#`	how many arguments were passed
`$@`	all the arguments supplied
`$$`	the process ID of the current script
`$?`	the exit status of the most recently run process

Input/Output streams

Shells use 3 standard I/O streams

- `stdin` is the standard input stream, which provides input to commands
- `stdout` is the standard output stream, which displays output from commands
- `stderr` is the standard error stream, which displays error output from commands

Shell has several **meta-characters** and **control operators**

Control operators

Character	Effect
`;`	Normal separator between commands
`&&`	Execute next command only if command succeeds
`||`	Execute next command only if command fails
`&`	Don't wait for result of command before starting next command
`|`	Use output of command as input for the next command
`> file_desc`	Send stdandard output of command to file descriptor
`< file_desc`	Use content of file descriptor as input

Redirections

Use the meta-character `>` in order to control the output streams `stdout` and `stderr` for a command or a bash script

From bash script

```
#!/bin/bash
#STDOUT to STDERR
echo "Redirect this STDOUT to STDERR" 1>&2
#STDERR to STDOUT
cat $1 2>&1
```

Output streams to file(s)

```
./my_script.sh > STDOUT.log 2> STDERR.err
```

How to Read a File Line By Line : input redirection

```
#!/bin/bash
# How to Read a File Line By Line

input="/path/to/txt/file"
while IFS= read -r line
do
  echo "$line"
done < "$input"
```

> by default `read` removes all leading and trailing whitespace characters such as spaces and tabs

Return codes

Linux command returns a status when it terminates normally or abnormally

- every Linux command has an exit status
- the exit status is an integer number
- a command which exits with a **0** status has **succeeded**
- a **non-zero** (1-255) exit status indicates **failure**

How do I display the exit status of shell command ?

```
date
echo $?
```

> List of special exit codes for GNU/Linux

How to store the exit status of the command in a shell variable ?

```
#!/bin/bash
date
status=$?
echo "The date command exit status : ${status}"
```

How to use the `&&` and `||` operators with **exit codes**

```
command && echo "success"
command || echo "failed"
command && echo "success" || echo "failed"
```

```
_files="$@"
[[ "$_files" == "" ]] && { echo "Usage: $0 file1.png file2.png"; exit 1; }
```

Hands-on exercise

1. In your `bash_exercises` folder, copy `exercise_3.sh` to `exercise_4.sh`
2. In this new file, loop over the words and write the number of letters of each word in a new file called `output.txt`
3. Now loop over the created file `output.txt` to get the total number of letters
4. Display the total run time and the total number of letters

Functions

- "small script within a script" that you may call multiple times
- great way to reuse code
- a function is most reuseable when it performs a single task
- good to put ancillary tasks within functions : logically separate from main code

```
#!/bin/bash
hello_world () {
    echo 'hello, world'
}
hello_world
```

Functions must be declared **before** they are used

> defining a function doesn't execute it

Variables Scope

```
# Define bash global variable
# This variable is global and can be used anywhere in this bash script
var="global variable"

function my_function {
# Define my_function local variable
# This variable is local to my_function only
echo $var
local var="local variable"
echo $var
}

echo $var
my_function
# Note the bash global variable did not change
# "local" is my_function reserved word
echo $var
```

Return Values

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

After completion, the return value is the **status** of the last statement (so 0-255)

It can also be specified manually by using `return` :

```
my_function () {
  echo "some result"
  return 55
}
my_function
echo $?
```

Return an arbitrary value (different from a return code) from a function :

- Assign the result of the function

```
my_function () {
  func_result="some result"
}
my_function
echo $func_result
```

- Better way is to send the value to `stdout` using `echo`

```
my_function () {
  local func_result="some result"
  echo "$func_result"
}
func_result="$(my_function)"
echo $func_result
```

Passing Arguments

In the same way than a bash script: see above (`$1` , `$*` , etc)

```
#!/bin/bash
print_something () {
    echo Hello $1
}
print_something Mars
```

> Athough it is possible, you should try to avoid having functions using the name of existing linux commands.

Hands-on exercise

1. Write a script called `exercise_5.sh` expecting **2 arguments**. If not exactly two arguments are provided:
   - Echo an error message
   - Exit with a non-zero error code
2. Write a function taking a **folder path** (e.g `/home/ucl/elic/xxxx`) and an **extension** (e.g `py`) as arguments
3. Use the `ls` command to list the files in the given path having with the given extension. Write this list to a file called `files_found.txt`.
4. Bonus : if there are no files, Exit with a non-zero error code

Shell vs Environment Variables

Consider the script `test.sh` below :

```
#!/bin/bash
echo "var1 = ${var1}"
echo "var2 = ${var2}"
```

Then run this script :

```
var1=23
export var2=12
bash test.sh
```

> By default, variables from the main interpreter are not available in scripts, unless you `export` them.

Subshells

- A subshell is a "child shell" spawned by the main shell ("parent shell")
- A subshell is a **separate** instance of the command process, run as a new process
- **Unlike calling a shell script** (slide before), subshells inherit the **same** variables as the original process
- A subshell allows you to execute commands within a separate shell environment = *Subshell Sandboxing*

  > useful to set temporary variables or change directories without affecting the parent shell's environment

- Subshells can be used for **parallel processing**

Syntax

A command list embedded **between parentheses** runs as a subshell :

```
#!/bin/bash
( command1 ; command2 ; command3 )
```

Or :

```
#!/bin/bash
bash -c "command1; command2; command3"
```

> Reminder : variables in a subshell are **not** visible outside the block of code in the subshell

Differences between Sourcing and Executing a script

- source a script = execution **in the current shell**

  > variables and functions are valid in the current shell after sourcing even if not `export`ed

- execute a script = execution in a new shell (in a subshell of the current shell)

  > all new variables and functions created by the script will only live in the subshell

Source a script using `source` or `.`

```
source myScript.sh
. myScript.sh
```

> official one is `.` Bash defined `source` as an alias to the `.`

Example

```
#!/bin/bash
COUNTRY="Belgium"
greeting() {
    echo "You're in $1"
}
greeting $COUNTRY
```

```
COUNTRY="France"
./myScript.sh # or bash or exec
echo $COUNTRY
greeting $COUNTRY # !!
```

```
COUNTRY="France"
source myScript.sh
echo $COUNTRY
greeting $COUNTRY
```

Debug

Tips and techniques for debugging and troubleshooting Bash scripts

use `set -x`

enables debugging mode : print each command that it executes to the terminal, preceded by a `+`

check the exit code

```
#!/bin/bash
if [ $? -ne 0 ]; then
    echo "Error occurred"
fi
```

use `echo`

Classical but useful technique : insert `echo` throughout your code to check variable content

```
#!/bin/bash
echo "Value of variable x is: $x"
```

use `set -e`

this option will cause Bash to exit with an error if any command in the script fails

Thank you for your attention

Running parallel processes in subshells

Processes may execute in parallel within different subshells

> permits breaking a complex task into subcomponents processed concurrently

Exemple : `job.sh`

```
#!/bin/bash
job() {
  i=0
  while [ $i -lt 10 ]; do
    echo "${i}: job $job_id"
    (( i++ ))
    sleep 0.2
  done
}
```

sequential processing (`manager_seq.sh`) or parallel processing (`manager_par.sh`)

```
#!/bin/bash
# manager_seq.sh
source job.sh
echo "start"
for job_id in {1..2}; do job ; done
echo "done"
```

```
#!/bin/bash
# manager_par.sh
source job.sh
echo "start"
for job_id in {1..2}; do job & done
wait # Don't execute the next command until subshells finish.
echo "done"
```

```
time ./manager_seq.sh
time ./manager_par.sh
```