Linux Basics IV: Basic shell scripting

Adriano Angelone, Graziano Giuliani

Course Outline

- UNIX/Linux Basics
- Intermediate shell commands
- Editing and compiling source code
- Text file manipulation
- **Basic shell scripting**

Download slides and exercise files with the command

```
git clone https://github.com/AA24KK/LinuxBasics.git
```

or download a ZIP archive at

```
https://github.com/AA24KK/LinuxBasics/archive/master.zip
```

Adriano: **aangelon@ictp.it**, Room 263, ICTP
Graziano: **ggiulian@ictp.it**

Why shell scripting

Reuse multiple times the same command:
less work, less bugs

What are the basics:

- Variables
- Conditionals
- Loops

Variables I

Variables contain data
they have a label (**name**) and a **content**

In bash, variables **have no type**:
everything is a string, with arithmetic sometimes possible

Variables II

Variables are initialized when you use them the first time:
you can call non-existing variables, usually trouble
(unless you put `set -u` in your script)

Assignment: give a variable a value

`<variable name>=<value>`

Expansion: access the value

`$<variable name>`

```
~ » cat example_script
#!/bin/bash

a="txt"
echo $a
-------------------------
~ » ./example_script
txt
```

Usually it's good to use `"$<variable name>"`
to avoid problems if variable contains spaces

Variables III

(Simple) printing: `echo`

Reading from standard input: `read`

`read -p`: **write prompt**

```
~ » cat example_script
#!/bin/bash

echo "hello"

read input
echo "$input"

read -p "please insert 2nd input :: " input2
echo "$input2"
```

```
~ » ./example_script
hello
123
123
please insert 2nd input :: 456
456
```

Easy to make variable values interact with strings

Use `${<variable name>}` to avoid expanding another variable

```
~ » cat example_script
#!/bin/bash

a="txt"
echo ${a}_stuff
```

```
~ » ./example_script
txt_stuff
```

Variables IV

Command substitution:

`<variable>=$(<command>)`

outputs command in variable

```
[aangelon@login02 ~]$ ls
arch  devil  entham  intel  lcmc  lcmc-test  lpmc  scripts
[aangelon@login02 ~]$ a=$(ls)
[aangelon@login02 ~]$ echo $a
arch devil entham intel lcmc lcmc-test lpmc scripts
[aangelon@login02 ~]$
```

Some variables are defined system-wide by default: e.g.,

- `HOME`: path to your home folder
- `PATH`: where commands will be searched for if no path specified

```
~ » echo $HOME
/home/nemesis3
-------------------------------------------------------------------------------------------------------------------------------
~ » echo $PATH
/home/nemesis3/.local:/opt/anaconda/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

Script arguments are variables:

- `$0`: script name
- `$1-...`: arguments
- `$@`: all arguments
- `$#`: argument number (+1, `$0` counts)

```
~ » cat example_script
#!/bin/bash

echo "Number of arguments = $#"
echo "Script name = $0"
echo "Argument 2 = $2"
echo "All arguments = $@"
-------------------------------------
~ » ./example_script a1 a2 a3 a4 a5
Number of arguments = 5
Script name = ./example_script
Argument 2 = a2
All arguments = a1 a2 a3 a4 a5
```

Conditionals I

Basic structure:

```
if <test> ; then <action 1> ; else <action 2> ; fi
```

Several types of conditions can be tested,
slightly different syntax

First type of condition:
test command output

Write command in condition
without parentheses

If return code = 0
(usually, success)
triggers `then`,
`else` otherwise

```
~ » cat example_script
#!/bin/bash

touch example_file

if mv example_file e
then
  echo 'It worked'
else
  echo 'It didnt work'
fi

# file not here anymore
if mv example_file e
then
  echo 'It worked'
else
  echo 'It didnt work'
fi
-------------------------------------------------------------
~ » ./example_script
It worked
mv: cannot stat 'example_file': No such file or directory
It didnt work
```

Conditionals II

Second type of condition:
primary expressions

- `[ -f <file> ]`: file exists
- `[ -d <directory> ]`: dir exists
- `[<string1> ==/!= <string2>]`
- `[<int1> <operator> <int2>]`

  `<operator>` can be:

  - `-eq/ne`: equal/not equal
  - `-lt/le`: less/less or equal
  - `-gt/ge`: greater/greater or equal

Conditions within `[...]`

`[... -a ...]` is AND,

`[... -o ...]` is OR,

`[! ...]` is NOT

```
#!/bin/bash

touch example_file

# This writes 'exists'
if [ -f example_file ]
then
  echo 'exists'
fi

a="txt1"
b="txt1"

# This writes 'equal'
if [ $a == $b ]
then
  echo 'equal'
fi

a=12
b=12
c=13

# This writes 'equal'
if [ $a -eq $b ]
then
  echo 'equal'
fi

#This writes 'a < c'
if [ $a -gt $c ]
then
  echo 'a > c'
else
  echo 'a < c'
fi
```

Iteration and loops

Perform actions several times:

```
for <variable> in <range> ; do <action> ; done
```

`<variable>` usually created on the spot

`<range>` can be:

- A given sequence (no commas)
- Matches to a regexp

`<action>` can use `<variable>`

Example:

cycle over command-line arguments:

```
for arg in "$@" ; ...
```

```
~/example_dir » ls
a.dat  b.dat  c.dat  example_script
-----------------------------------
~/example_dir » cat example_script
#!/bin/bash

for num in 1 2 3
do
  echo $num
done

for file in *.dat
do
  echo $file
done
-----------------------------------
~/example_dir » ./example_script
1
2
3
a.dat
b.dat
c.dat
```

Exercise: file processing

This will be a long exercise:
complex programs are done in steps
(start simple, then add complexity)

We have an external program, `binaver`, to average a column of a file:

```
binaver -r<rows> -c<columns> -k<column to average> <file>
```

Requires information about the files, only accepts one at a time

We will write a script (**wrapper**) which:

- Counts automatically rows and columns
- Excludes commented lines
- Checks the integrity of the file
- Applies `binaver` to multiple files

Step I: counting rows

Rows can be counted with the `wc -l <file>` command:
the command however outputs `<line number> <file>`

Task 1:
Write a script which assigns to a variable only the row number of a file
(hint: use command substitution and a pipeline with `awk` or `cut`)

Assume the file to be the first command line argument
(remember how you access it)

Step II: counting columns

Task 2:
Extend the previous script, assigning to a variable
the number of columns of the same file
(hint: one way uses `awk`, its internal variables, and `sort`)
For now, assume that all rows have the same number of columns

Task 3:
Modify the script so that it proceeds only if all rows
have the same number of columns
(hint: if you did it as suggested before,
you could use the previous bit of the script here)
This checks the **integrity** of the file

Step III: comments and finalization

Task 4:
Modify all parts of the previous script to exclude commented lines
(starting with `#`)
(hint: you can use `grep`)

Task 5:
Ask the user which column he wants to perform the average on
(hint: `read -p`)

Task 6:
Complete the script, applying `binaver` to `$1` with the known info

Step IV: robustness, multiple files

Task 7:
Extend the script to accept multiple files and act on all of them
(hint: loop over arguments, `binaver` takes one at a time)

Task 8:
Increase robustness: skip a file if it doesn't exist
(hint: use conditionals, the `else` branch can be empty)

We're done!

Final Notes

A few pieces of advice:

- Sometimes it's worth settling for the easy and tedious solution. However, try to check out if you can do things in a smarter way and **work less (!!!)**
- You will remember the details in time. For now, **rough is enough**: "*I can do this with* `tail`*, but I don't remember the options*" is ok, if you have internet access
- Give `vim`/`emacs` and the shell a shot, it will pay you back tenfold

Good luck out there!