ICTP DP Linux Basic Course - Shell Script

ESP Students - First Semester

Graziano Giuliani
ggiulian@ictp.it

The Abdus Salam International Centre for Theoretical Physics

ICTP Diploma Program
September 5, 2023

Course Outline [1]

Daily program

- **UNIX/Linux**
- **Programming on Linux**
- **Text file manipulation**
- **Basic BASH and Python**
  1. BASH variables
  2. BASH programming
  3. Python programming

Slides:

`http://tinyurl.com/2jsvfbd6`

or the LaTeX source on GitHub:

`https://github.com/graziano-giuliani/LinuxBasics`

[1] Course created in 2019 with Adriano Angelone, now LPTMC-FR

Shell Scripting

From REPL to Programming Language

Why Shell Scripting

Reusing multiple times the same command sequence?
Write the sequence in a script and create a new command.

Elements of a Programming Language

- **Execution flow**: Entry point and return point
- **Input/Output**: Interaction
- **Variables**: Data representation and storage
- **Conditionals**: Branch execution on condition on variables
- **Loops**: Repeat code until condition on variables met

Bash shell is a complete programming language.

Variables

Name and Value

Variables contain data. They have a label (**Name**) and a content (**Value**).
In bash, variables **have no type**

- Everything is a string
- Arithmetic sometimes possible on integer values

Variables

Assign value to name and expand name into value

Assignment

Use the `=` character between the variable name and its value content.

- `name="value"`
- `message="A whole string"`
- `number=3`

Expansion

Prepend the `$` character to the variable name.

- ```
  b=$number
  c=$b
  ```
- ```
  part1="Hello"
  part2="World"
  whole="${part1} ${part2}!"
  ```

Basic I/O

echo, read and how to catch program output

print

Use the `echo` command.

```
message="Welcome to my program!"
result=42
echo $message: The answer is $number
```

read

Use the read command, eventually using the `-p` (prompt) option.

```
read -p "Please enter your name: " user_name
read -p "Please enter your Age : " user_age
echo ${user_name} age is ${user_age}.
```

Subshell

Output from programs can be stored in variables:

```
listing=$(ls)
echo $listing
```

Variables

Shell environment variables

Every shell has built-in variables assigned at startup.
Their names and values can be listed using the `env` command.

Environment variables

- `USER` : User name
- `HOME` : User home folder path
- `PATH` : Search path for user binary programs

To set an environment variable, use `export`:

```
export LC_ALL="C"
```

Try the `env` program in your shell to see the environment variables.
Can you sort their values by name?
Try changing the value of `LC_ALL` to the value `"C"` and run `ls`.
Set it back to the value `"en_US.UTF-8"`. Run `ls` again.
Examine what changes.

Variables

Command line arguments as variables in script

Script arguments are variables

Scripts

```
#!/bin/bash
echo "Script name = $0"
echo "Number of arguments = $#"
echo "First argument = $1"
echo "Second argument = $2"
echo "All arguments = $@"
```

Create a simple script with the above content and run it:

```
bash test_script.sh a1 a2 a3 a4 a5
```

Can you foretell the output?

Conditionals

Basic structure: test command success

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

Test command result output

```
~ » 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
```

True is `0`

False is any value not `0`

Conditionals

test conditions on file and variables

- `[ -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
- `[... -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

Perform actions several times

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

- `variable` is created for the loop and cannot be reached outside of the loop. `<action>` can use `<variable>`
- `range` is a sequence or a regexp to be exapnded

```
~/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

Put it all together: File processing

This will be a long exercise

Complex programs are done in steps: start simple, then add complexity.

Assume that we have an external program, `binaver`, to be used to average data of a single targer column from a column based file.
It requires information about the file, its row and column total number, and only accepts one file at a time.
The program can be run with the following syntax:

```
binaver -r<rows> -c<columns> -k<target_column> <file>
```

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

Counting rows

Task 1

Write a script which assigns to a variable the number of rows of a file.

Rows can be counted with the `wc -l <file>` command, but the command however outputs `<line number> <file>`
Rows can also be counted using `awk`

Hint

Use command substitution and a pipeline with `awk` or `wc` and `cut`.
Assume the file to be the first command line argument and remember how you access it.

Solution

Counting rows

Using `awk`

```
#!/bin/bash

file=$1
nrows=$(awk 'END{print NR}' $file)
# Debug
echo "File $file has $nrows lines"
```

Using `wc` and `cut`

```
#!/bin/bash

file=$1
nrows=$(wc -l $file | cut -d " " -f 1)
# Debug
echo "File $file has $nrows lines"
```

Step

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

Solution

Counting columns

Rows and columns both

```
#!/bin/bash

file=$1
nrows=$(awk 'END{print NR}' $file)
ncols=$(awk '{print NF}' $file | sort -u)
if [ $(echo $ncols | awk 'END{print NF}') -ne 1 ]
then
  echo "File lines do not have the same number of columns"
  exit 1
fi
# Debug
echo "File $file has $nrows lines and $ncols columns"
```

Step

Exclude 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

Solution

Exclude comments and finalization

Single file solution

```
#!/bin/bash

file=$1
nrows=$(grep -v "^#" $file | awk 'END{print NR}')
ncols=$(grep -v "^#" $file | awk '{print NF}' | sort -u)
if [ $(echo $ncols | awk 'END{print NF}') -ne 1 ]
then
  echo "File lines do not have the same number of columns"
  exit 1
fi
read -p "Enter column to be processed (1-$ncols):" target
# Debug
echo "binaver -r$nrows -c$ncols -k$target $file"
```

Step

Multiple files, robustness

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

Solution

Multiple files, robustness

Final complete solution

```
#!/bin/bash

for file in $@
do
  if [ -f $file ]
  then
    nrows=$(grep -v "^#" $file | awk 'END{print NR}')
    ncols=$(grep -v "^#" $file | awk '{print NF}' | sort -u)
    if [ $(echo $ncols | awk 'END{print NF}') -eq 1 ]
    then
      echo "File $file fit for processing."
      read -p "Enter column to be processed (1-$ncols):" target
      # Debug
      echo "binaver -r$nrows -c$ncols -k$target $file"
    fi
  fi
done
```

Python

Whet your appetite with example!

Read a dataset and plot

We want to read a scientific dataset, stored on disk in a scientific format data file, and plot the content.

- Access through community maintained API

Environment

```
source /home/netapp-clima/users/ggiulian/m19.sh
```

NetCDF Data format

xarray: labelled multi-dimensional arrays

NetCDF Scientific data format

Standard for the climate and forecast is the netCDF.

- Use the xarray package

python code

```
#!/usr/bin/env python3

from matplotlib import pyplot as plt
import xarray as xr

disk = "/home/esp-shared-a"
datapath = "Observations/BERKELEYEARTH/Gridded"
datafile = "Complete_TAVG_Daily_LatLong1_2010.nc"
ncfile = xr.open_dataset(disk+'/'+datapath+'/'+datafile)
ncfile.temperature
data = ncfile.temperature[0,:,:]
data.plot( )
plt.show( )
```

Seismic Data format

obspy : Python framework for processing seismological data

obspy

Provide parsers for common file formats, clients to access data centers and seismological signal processing routines which allow the manipulation of seismological time series

python code

```
#!/usr/bin/env python3

from matplotlib import pyplot as plt
import obspy

site = "https://github.com/obspy/examples/raw/master"
file = "IU_ULN_2015-07-18T02.mseed"
data = obspy.read(site+'/'+file)
data.plot( )
plt.show( )
```