Bash and **Git**

LHCb **Starterkit** 2024

Lecturers:
Dr. Uzziel Perez and Dr. Mindaugas Sarpis

What is Bash?

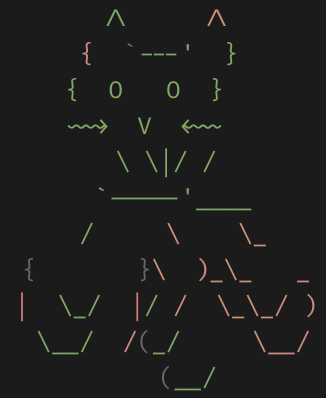
- Bash = **Bourne Again shell**
- Bash is a shell program designed to listen to commands
  - *zsh*, *csh* are other shell programs
- On Windows install **Windows Subsystem for Linux** (WSL)

```
wsl --install
```

- On macOS/Linux open terminal and you have a bash shell.

Why bother???

Learning bash is a powerful way to automate tasks on linux and boosts productivity. It will also help you finish your PhD faster XD ...



Checking the manual

Let's get our hands dirty… Open the terminal and let's get right on to it!

First command to learn:

```
man bash
```

This shows the documentation on Bash including all the options that can be used with this command.

```
NAME
       bash - GNU Bourne-Again SHell

SYNOPSIS
       bash [options] [command_string | file]

COPYRIGHT
       Bash is Copyright (C) 1989-2020 by the Free Software

DESCRIPTION
       Bash is an sh-compatible command language interpreter
       ful features from the Korn and C shells (ksh and csh)

OPTIONS
       All  of  the  single-character  shell  options docume
       shell is invoked.  In addition, bash interprets the f

       -c        If the -c option is present, then commands
                 command_string,  the first argument is assi
                 to $0 sets the name of the shell, which is
       -i        If the -i option is present, the shell is i
```

SSH: Connecting to a remote computer

```
man ssh
```

SSH or Secure Shell is a protocol used to securely connect to a remote computer or server over an unsecured network.

SSH: Connecting to a remote computer

```
man ssh
```

SSH or Secure Shell is a protocol used to securely connect to a remote computer or server over an unsecured network.

Let's check if you could ssh into lxplus.

```
ssh -X USERNAME@lxplus.cern.ch
```

While it's not necessary that we work on lxplus this morning, better do the prerequisites for the other lessons throughout the week.

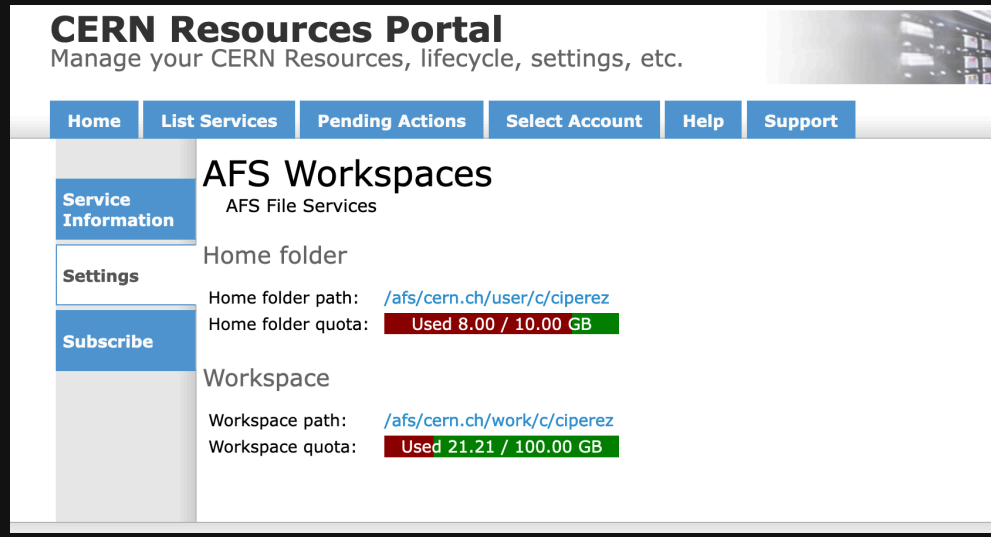
What is Lxplus?

Lxplus is CERN's interactive Linux service for all users.

- Provided by the **IT Department**.

How to activate AFS workspaces:

1. Go to the CERN Resources Portal.
2. Navigate to **List Services → AFS Workspaces → Settings**.



SSH to LXPLUS

```
ssh -X username@lxplus.cern.ch
```

The `-X` flag enables basic X11 forwarding (running GUI).

Accessing the Grid

To access the grid, we need to initialize a valid **Grid Proxy Certificate** which is essential for accessing various LHCb and CERN computing resources (data storages, job submission and file transfer).

```
lhcb-proxy-init
```

This command is a wrapper around the standard `voms-proxy-init`. You should see a similar output as on the right.

```
ciperez:~$ lhcb-proxy-init
Generating proxy ...
Enter Certificate password: **********
Added VOMS attribute /lhcb/Role=user
Uploading proxy ..
Proxy generated:
subject      : /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=
issuer       : /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=
identity     : /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=
timeleft     : 23:53:59
DIRAC group  : lhcb_user
path         : /tmp/x509up_u81686
username     : ciperez
properties   : NormalUser, PrivateLimitedDelegation
VOMS         : True
VOMS fqan    : ['/lhcb/Role=user']

Proxies uploaded:
 DN
 /DC=ch/DC=cern/OU=Organic Units/OU=Users/CN=ciperez/CN=773
```

Download Tutorial Pack

```
wget -O data-shell.zip https://cern.ch/go/9rKZ && unzip data-shell.zip && rm data-shell.zip
```

or for this tutorial see:

```
/afs/cern.ch/user/c/ciperez/public/data-shell
```

For additional materials see:

```
/afs/cern.ch/user/c/ciperez/public/bash_practice
```

```
                  ___________
                 /           \
        e⁻   →→→ |     *     | ←←←   e⁺
  →→→→→→→→→→→→→→|    /\/\    |←←←←←←←←←←←←←←
B             →→|   /  X  \   |←←             B
  →→→→→→→→→→→→→→|    \/\/    |←←←←←←←←←←←←←←
        e⁻   →→→ |     *     | ←←←   e⁺
                 \___________/
```

File System navigation

Commands to navigate the file system. If you are using `lxplus`, you are most-likely using the **AFS** or the *Andrew File System* which is a distributed file system where multiple computers are allowed to share files and data efficiently.

Command	Description
pwd	Lists the path to the working directory
ls	List directory contents
ls -a	List contents including hidden files (Files that begin with a dot)
ls -l	List contents with more info including permissions (long listing)
ls -r	List contents reverse order
cd	Change directory to home
cd [dirname]	Change directory to specific directory
cd ~	Change to home directory
cd ..	Change to parent directory
cd -	Change to previous directory (which could be different than the parent of course
find [dirtosearch] -name [filename]	Find location of a program

One can also group flags together like `ls -la`. Credits to *bradtraversy* for this slide.

Modifying files and directories

Below are a list of commands to modify files and directories.

Command	Description	Examples
mkdir [dirname]	Make directory	mkdir starterkit24
touch [filename]	Create file	touch scratch.py
rm [filename]	Remove file	rm scratch.py
rm -i [filename]	Remove directory, but ask before	rm -i scratch.py
rm -r [dirname]	Remove directory	rm -r startkerkit24
rm -rf [dirname]	Remove directory with contents	.....
rm ./*	Remove everything in the current folder	
cp [filename] [dirname]	Copy file	
mv [filename] [dirname]	Move file	
mv [dirname] [dirname]	Move directory	
mv [filename] [filename]	Rename file or folder	
mv [filename] [filename] -v	Rename Verbose - print source/destination directory	

Credits to *bradtraversy* for this slide..

Some extra tips!

- We can also do multiple commands at once with the `&&` operator.

```
mkdir starterkit && cd startkerkit
```

- History

```
history : to print out entire history
Ctrl + r: search command history
!n : prints out the nth command in the history

# Keyboard shortcuts
- Up Arrow: Will show your last command
- Down Arrow: Will show your next command
- Tab: Will auto-complete your command
```

- Keyboard shortcuts are cool

```
- ### Keyboard Commands
- clear: Will clear the screen
- Ctrl + C: Will cancel a command
- Ctrl + R: Will search for a command
- Ctrl + D: Will exit the terminal

- ### Cursor
- Ctrl + A: Go to the beginning of the command line
- Ctrl + E: Go to the end of the command line
- Ctrl + B: Move back one character
- Ctrl + F: Move forward one character
- alt + right: Move cursor forward one word
- alt + left: Move cursor back one word
```

Listening Break

Try out the commands you learned in the exercises from Analysis Essentials: Working with Files and Directories.

We also want to set the following environment variables:

```
export bash_data=/Users/uzzielperez/data-shell # or your path to data-shell
echo $bash_data
# Similaryly
export bash_practice=/Users/uzzielperez/Desktop/bash_practice # or your full path
```

For persistence, you can also set environment varialbes in your `~/.bashrc` file.

Display and Redirection

- To display messages

```
echo "Hello, My name's Forrest."
```

- To create a file with Echo

```
echo "Hello, My name's Forrest." > helloworld.txt
```

- To append to a file

```
echo "Forrest Gump." >> helloworld.txt
```

- To display the content of the file

```
cat helloworld
```

In general the right angle bracket tells the system to output results into a target.

```
echo "  /\___/\ "
echo " (  o o  )"
echo " (  =^=  )"
echo "  --m-m-- "
```

To save the cat into a file:

```
echo -e " /\\___/\\\n(  o o  )\n(  =^=  )\n (--m-m--)" > ca
```

Here's a dog:

```
echo -e "  _________________
< Hello, nice to meet you! >
  _________________
     \\   ^__^
      \\  (oo)\\_______
         (__)\\       )\\/\\
             ||----w |
             ||     || "
```

Redirection, Pipes and Filters

Highlighting a few things from the tutorial.

```
cd data-shell && cd molecules
wc *.pdb # counts lines, words, chars in files
wc -l *.pdb # outputs only the number of lines
wc -l *.pdb > lengths.txt # redirects output to a file
cat lengths.txt # concatenate - prints file content
sort -n lengths.txt # sorts out the result
head -n 1 lengths.txt # prints out the first line
tail -n 3 lengths.txt # prints out the last 3 lines
```

A vertical bar between the two commands = **pipe**.

```
Command1                Command2
 _______                _______
|       |     pipe     |       |
| wc-l  | ————————+——→| sort  |
|_______|     |        |_______|
            data
            flow

Output of wc -l ——→ Input for sort
e.g.

wc -l *.pdb | sort -n
wc -l *.pdb | sort -n | head -n 1
```

More commands

Below are a list of commands to modify files and directories.

```
| Command                              | Description                          | Examples                                |
| ————————————————————————————————————| ————————————————————————————————————| ————————————————————————————————————— |
| grep [pattern][file]                 | Looks for a pattern in file          | grep "exhaust-port" rebel_intel.txt     |
| find [directory] -n[name]            | Finds a file in directory            | find . -n rebel_intel.txt               |
```

- `grep` is short for *global regular expression print*. It is a useful command to search for matching patterns in a file.
- `find` is for finding file/s in directories

Regular Expressions with grep

Highlighting some parts of the Analysis Essentials:

```
cd $bash_data # an environment var we set earlier to the /path/to/data-shell
cd writing
cat haiku.txt
grep the haiku.txt # find the pattern "the" in haiku.txt
grep -i the haiku.txt # find the pattern "the" (case-insensitive) in haiku.txt
grep -w The haiku.txt  # find the word "The" in haiku.txt
grep --color='auto' 'the' haiku.txt
```

Let's try some Regex:

```
cd $bash_practice
grep --color='auto' "unix" geekfile.txt # ^ start of the line
grep --color='auto' "^unix" geekfile.txt # ^ start of the line
grep --color='auto' "unix$" geekfile.txt # $ end of the line
grep --color='auto' 'os\.' geekfile.txt  # match `os`` before a period
grep --color='auto' 'os[.[:space:]]*$' geekfile.txt # match `os` regardless of punctutation
```

For a full cheatsheet see this and this tutorial.

More commands

Below are a list of commands to modify files and directories.

```
| Command                                | Description                                | Examples                                  |
| ———————————————————————————————————————| ———————————————————————————————————————| ————————————————————————————————————————— |
| sed "s/[find]/[replace]/g" [file]      | Find and replace a pattern in a text       | sed "s/Luke/Leia/g" rebel_intel.txt       |
| ln -s [filename] [symlinkname]         | Create a symlink                           | ln -s [rebel_intel.txt][rebel.txt]        |
```

- `sed` stands for *stream editor* and it can be used to edit text files. It is commonly used to replace occurences of words in a file.
- Creating a symlink is neat way to create shortcut to the original file without having to copy the file.

Other Comands

Command	Description
ps	a.k.a. process status displays all processes
ps aux	display all processes in the system, of aux flags.
chmod	change file modes
chmod u+x	give yourself (owner only) permission to execute a file you own
chmod +x	Adds permission to execute a file = `chmod a+x`
chown	change file onwer or group
top	display sorted information about processes
kill <pid>	terminate or kill a signal process
kill -9	non-ignorable kill!
lsof +D	list open files. Useful when prematurely killing a process
tar	manipulate tape archives
zip	package and compresss archive files

Important commands like `du` (disk usage) and `df` (remaining free space) are also found in the Analysis Essentials.

Cat and `sed` Exercise

```
cat LHCb.txt
```

You find that the `Belle` experiment was mistakenly written instead of LHCb!

You can use the `sed` command to replace ALL the instances of `Belle` within the txt.

```
sed 's/Belle/LHCb/g' LHCb.txt
```

If you only want to replace the nth occurence of a pattern in a line:

```
$sed 's/unix/linux/2' geekfile.txt # here n = 2, you can al
```

Store this cat into a `txt` file.

```
cat cat.txt
```

This will print out

```
 /\_/\
( o o )
 > ^ <
```

```
# Use sed to replace open eyes (o) with closed eyes (X)
sed "s/o/X/g" $cat_file > sleeping_cat.txt
echo "Cat with eyes closed has been written to sleeping_cat
cat sleeping_cat.txt


 /\_/\
( X X )
 > ^ <
```

Loops, Conditionals, Arrays and shell scripts

Listening Break

Try out the Loops exercises from the Analysis Essentials: Working with Files and Directories.

Here are some highlights:

```
$ cd /Users/uzzielperez/data-shell/creatures
$ for filename in basilisk.dat unicorn.dat
do
   head -n 3 $filename
   # Or do some other thing here like echo the filename
done
```

Why Shell Scripting?

One can string together various pieces of their analysis and save time…

```
export analysis_dir=$HOME/work/analysis
alias mainscript="python3 main.py"
lhcb-proxy-init
source setupLCG.sh
function run_analysis(){
  python3 do_Fit.py
  python3 calc_eff.py
  python3 main_analysis.py
  python3 plot_results.py
}
```

In general, it also helps with tedious and repetitive tasks.

Bash Scripting Crash Course

Open a file like `vi starwars.sh` and put these lines:

```
#!/bin/bash
name ="Luke Skywalker"
echo "Hello, $name"
```

To run the script:

```
bash starwars.sh
```

One could also do:

```
chmod u+x starwars.sh
./starwars.sh
```

- User input

```
read -p "Would you like to look at MC or DATA: " datatype
echo "Hello $datatype"

read -p "Enter data-taking year: " year
echo "You are analyzing $datatype $year "
```

- Arguments

`$1, $2 ..` store the arguments passed to the script…

```
echo $0
echo $1
echo $2
echo "${@}" # Access all the arguments [More on this later]
```

So if you do `./script.sh condition1 condition2`, what happens? It just echos the strings passed on to the script.

Bash Scripting Crash Course

- [[]] enables to use operators.
- Comparisons: `=, ≠, >, <, ≤, ≥`

```
if [[ "$name" == "dorothy" ]]
then
   echo "hi dorothy we missed you"
else
   echo "welcome $name"
fi
```

Inspired by the Missing Shell Scripting Crash Course

- Test commands for some complex operations

```
# Compare Strings
[[ "$str1" == "$str2" ]]
[[ "$str1" ≠ "$str2" ]]

# Integer Comparisions
[[ "$int1" -eq "$int2" ]] # $int1 == $int2
[[ "$int1" -ne "$int2" ]] # $int1 ≠ $int2
[[ "$int1" -gt "$int2" ]] # $int1 > $int2
[[ "$int1" -lt "$int2" ]] # $int1 < $int2
[[ "$int1" -ge "$int2" ]] # $int1 ⩾ $int2
[[ "$int1" -le "$int2" ]] # $int1 ⩽ $int2

# And or
[[ ... ]] && [[ ... ]] # And
[[ ... ]] || [[ ... ]] # Or
```

Minimal Safe Bash Script Template

A tutorial I wish I had when I was younger is from Bash Script Template

- FAIL FAST

```
#!/usr/bin/env bash
cp important_file ./backups/
rm important_file
```

Suppose the backups directory does not exist. If there is no safety option `set -Eeuo pipefail`, bash jumps into the next command and deletes the important file before you can react. This line configures the shell to exit immediately on any error...

- Get the Location

```
script_dir=$(cd "$(dirname "${BASH_SOURCE[0]}")" &>/dev/null && pwd -P)
```

Often we find the scripts we need to run in some other directory e.g. `/some/long/path/to/script.sh`. This can be fixed by going to the directory before execution with `cd /some/long/path/to/ && ./script.sh`

Bash Profile and persistency settings

The `~/.bash_profile` is used for defining user settings for a login shell.

```
# Load .bashrc if it exists
test -f ~/.bashrc && source ~/.bashrc

if [-f ~/.bashrc]; then
  . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
echo "$(date + [%F_%H:%M]) at $(hostname)" >> .lxnodes # us
export PATH=$HOME/.cargo/bin:$PATH"

# Enable text color and formatting
export PS1="\[\033[36m\]\u:\[\033[33m\]\w\[\033[m\]\$ "
export CLICOLOR=true
```

The `~/.bashrc` file provides a place where you can set up variables, functions and aliases and helps reduce redundant effort.

```
# This is where you put your hand rolled scripts (remember
PATH="$HOME/bin:$PATH"

alias ll ='ls -l -h'
alias la='ls -a -l -h'

function mcd (){
  mkdir $1; cd $1
}

alias eosuser='cd /eos/user/c/ciperez'
alias afsdir='cd /afs/cern.ch/work/c/ciperez
```

Ref: Some useful things to add to `~/.bashrc

Miscellaneous

- TMUX, Screen

Lets you split session into windows and also lets you log out and having the session running. For big workloads, better to use HTCondor. Helps with monitoring CPU/memory usage too.

See a Quick and Easy Guide to TMUX and How to Use Linux Screen.

- LCG Stacks and Apptainer

If you want an environment where everything works harmoniously, you might want to create a conda environment, or a python environment.

It is however better to rely on the already installed software to work with the platform you currently have.

```
source /cvmfs/sft.cern.ch/lcg/views/setupViews.sh <LCG_number> <platform>
```

For ML related stuff, one can also get gpu-supported programs such as `tensorflow` with `LCG_106cuda for lxplus-gpu`. To check the latest LCG releases click here.

THE END!

This is just a simulation.

Backup

Bash Scripting Crash Course

Inspired by the Missing Shell Scripting Crash Course

Variable

Assigning value to a variable needs `$`, otherwise bash will treat name as a string literal and it will output `Hello name` instead.

```
#!/bin/bash
name ="Luke Skywalker"
echo "Hello, $name"
```

To run,

```
$user chmod
$user ./script.sh
```

User input

```
read -p "What is you name: " name
echo "Hello $name"

read -p "Enter an action: " verb
echo "You are ${verb}ing"
```

- ## Arguments

`$1, $2 ..` store the arguments passed to the script…

```
echo $0
echo $1
echo $2
echo "${@}" # Access all the arguments [More on this later]
```

So if you do `./script.sh condition1 condition2`, what happens? It just echos the strings passed on to the script.

Bash Scripting Crash Course

Inspired by the Missing Shell Scripting Crash Course

Logical Comparisons

- [[]] enables to use operators.
- Comparisons: `=, ≠, >, <, ≤, ≥`
- Leave some space on both ends of brackets… :)

```
if [[ "$name" == "adam driver" ]]
then
   echo "hi adam we missed you"
else
   echo "welcome $name"
fi
```

Test commands for some complex operations

```
[[ -e "$file" ]] # True if file exists
[[ -d "$file" ]] # True if file exists and is a directory
[[ -f "$file" ]] # True if file exists and is a regular fil
[[ -z "$str" ]]  # True if string is of length zero
[[ -n "$str" ]]  # True is string is not of length zero

# Compare Strings
[[ "$str1" == "$str2" ]]
[[ "$str1" ≠ "$str2" ]]

# Integer Comparisions
[[ "$int1" -eq "$int2" ]] # $int1 == $int2
[[ "$int1" -ne "$int2" ]] # $int1 ≠ $int2
[[ "$int1" -gt "$int2" ]] # $int1 > $int2
[[ "$int1" -lt "$int2" ]] # $int1 < $int2
[[ "$int1" -ge "$int2" ]] # $int1 ⩾ $int2
[[ "$int1" -le "$int2" ]] # $int1 ⩽ $int2

# And or
[[ ... ]] && [[ ... ]] # And
[[ ... ]] || [[ ... ]] # Or
```

Bash Scripting Crash Course

Inspired by the Missing Shell Scripting Crash Course

- ## Arrays and Functions

```
arr=(a b c d)
```

To read:

```
echo "${arr[1]}"     # Single element
echo "${arr[-1]}"    # Last element
echo "${arr[@]:1}"   # Elements from 1
echo "${arr[@]:1:3}" # Elements from 1 to 3
```

To insert:

```
arr[5]=e                          # direct address and in
arr=(${arr[@]:0:1} new ${arr[@]:1}) # Adding 'new' to array
```

To delete:

- ## Deleting needs re-indexing

```
arr=(a b c d)
unset arr[1]
arr=("${arr[@]}")
echo << "${arr[1]}" # c
```

- ## Functions

```
greet() {
  echo "Hello, $1"
}

greet Bash # Hello, Bash
```

Shell Scripts

Write a new script called `my_script.sh` with your favorite editor.

- *Loops* and *if statements* need a ";"

```
j = 20
for i in {0..10};
  do
    echo $i
    (( j+= 1))
  done

if [-f $HOME/.bashrc ];
  then
    echo Have .bashrc
fi
```

- Variables Assigning value to a variable needs `$`

```
a = $((j+2))
echo $a, $j
```

We can add some safety options at the top of the script.

```
#!/usr/bin/env bash

# my_script.sh

# Safety options
# -u :Undefined variables are treated as errors
# script will stop when encountered
# -e: if any commands in the script fail
# the script immediately fails
# -o: pipefail prevents the script from running in pipes

set -eux -o pipefail
shopt -s expand_aliases

j = 20
for i in {0..10};
  do
    echo $i
    (( j+= 1))
  done
....
```

Shell Scripts

Write a new script called `my_script.sh` with your favorite editor.

- *Loops* and *if statements* need a ";"

```
j = 20
for i in {0..10};
  do
    echo $i
    (( j+= 1))
  done

if [-f $HOME/.bashrc ];
  then
    echo Have .bashrc
fi
```

- Variables Assigning value to a variable needs `$`

```
a = $((j+2))
echo $a, $j
```

We can add some safety options at the top of the script.

```
#!/usr/bin/env bash

# my_script.sh

# Safety options
# -u :Undefined variables are treated as errors
# script will stop when encountered
# -e: if any commands in the script fail
# the script immediately fails
# -o: pipefail prevents the script from running in pipes

set -eux -o pipefail
shopt -s expand_aliases

j = 20
for i in {0..10};
  do
    echo $i
    (( j+= 1))
  done
....
```