BASH 101

Jacques Bourg

October 23, 2024

ABSTRACT

I give a brief introduction to Bash to help you get started quickly.

Keywords Bash, Linux, Shell, Command-line interface, Terminal, Scripting, Automation.

1 Introduction to Bash

According to wikipedia a shell is a computer program that exposes an operating system's services to a human user or other program. Bourne Again Shell or Bash is the Unix shell from the GNU project. It is free, under GPL license.

2 Commands

Command	Description	Example
cd	change directory	$cd\ ./data$
mkdir	create directory	mkdir data
ls	list files and directories	ls
cp	copy files (source -> destination)	cp ./data/source.txt ./data/destination.txt
mv	move or rename files	mv ./data/source.txt ./data/destination.txt
rm	rm file (-r remove folders recursively)	rm oldfile.txt
cat	concatenate files and print to standard output	cat file1.txt file2.txt > file12combined.txt
head	print the head of a file	head -n 5 data.csv (first 5 lines)
tail	print the tail of a file	tail -n 5 data.csv (first 5 lines)
grep	look for pattern in a file, return line	grep 1959 rains.txt: 1964, 10
sed	find and replace strings in file	$sed\ 's/old\_string/new\_string/g'\ data.csv$
awk	text processing command	awk 'print $1, $3' data.csv (1st and 3rd fields of each line)
echo	print text to standard output	echo "Hello, world!"
pwd	print current directory	
env	list environment variables	
ps	list active processes	
top	list of processes sorted by cpu usage	

Command	Description	Example
which	outputs the full path of the command	which ls
curl	send a get request	curl -O https://example.com/image.jpg
sudo	allows executing a command as a super-ruser	sudo mkdir new_directory
history	prints past commands	
clear	clear the screen	
find	finds the files which given characteristics	find . -name "file.txt"
man	user manual of a command	man ls
kill	kill a process	kill 1234 (1234 is the process id)
expr	basic arithmetic in the shell	expr 1 + 3
bc	basic calculator	echo "scale = 3; 5 / 4" | bc
exit	close the terminal	
reboot	reboot the system	
pip	install packages fromPyPI	pip install numpy
ssh	connect to serves as an user	ssh -X user@example.com

3 Pipes

Pipes are a way to compose instructions in bash. One can compose 3 commands like this:

```
command1 | command2 | command3
```

The output of command1 is the input of command2 and the output of command2 is the input of command3.

For instance:

```
ls | wc -l > file.txt
```

This lists the files and folders in the actual directory, this creates a list, and then we count the lines in the previous list, and outputs the number of files in a file called file.txt. In summary, this command counts the number of files and folders and saves this number in a file.

Example of common pipes in bash:

```
grep 'Cosette' lesmiserables.txt | cat | wc -l
```

This command counts the number of lines in the file "lesmiserables.txt" that contain the word "Cosette".

```
cat file.txt | grep "banana"
```

This command outputs the lines contained in file.txt containing the word banana.

```
cat file.txt | grep "banana" | wc -l
```

This command counts the number of lines containing the word banana in the file file.txt

```
cat file.txt | grep "banana" | sed 's/banana/apple/'
```

```
cat file.txt | sort | uniq
```

Sort and removes duplicates.

```
ls -l | awk '{print $9"\t"$5}' | column -t
```

Formats the output of ls -l into a table.

```
ls | wc -l > file_count.txt
```

creates a file with the count of the number of files.

```
ls | wc -l >> file_count.txt
```

This appends to an existing file the result of ls.

3.1 Regex usage in Bash

Several commands like grep, sed, awk and find use regular expressions for pattern matching.

```
> grep -E "Cos[aeiou]tte" lesmiserables.txt

> sed -E 's/^Error [0-9]+/Warning/' file.txt
```

Replace error by warning.

```
> awk '/pattern/ {print}' file.txt
```

Print lines that match a pattern

```
> find . -regex '.*\.txt' # Finds all .txt files
```

4 Variables and arrays

4.1 Variables

We assign string variables by using = sign without space, we call them using $ symbol. In the terminal:

```
> name="Alice"
> echo $name
```

For numerical variables:

```
> (num=14)
> echo $num
```

4.2 Arrays

As in python, bash uses zero indexing. In the following examples, look at the spaces between the declared array and the equal and the array, there are no commas neither.

4.2.1 Numeric and string arrays

```
array_1=(1 2 4)
array_2=(a b c)
array_3=("Hello friends" b c)
array_4=(a b c 4)
```

In example 1 and 2 we declare respectively a numeric array and a string array. In example 3, we declare a string array, and we use double quotes since there are special characters (like space). In example 4, 3 is declared as a string.

We can declare iteratively an array like this:

```
fruits=()
fruits[0]="apple"
fruits[1]="banana"
fruits[2]="orange"
```

As we see, bash uses 0 based indexing. Like in python fruits[-1] gives the last element of the array.

To access all the elements of an array, use the command @. In for loop one writes:

```
for fruit in "${fruits[@]}"; do
  echo "$fruit"
done
```

To know the length of an array:

```
length=${#fruits[@]}
```

One can do slicing:

```
for fruit in "${fruits[@]:0:2}"; do
  echo "$fruit"
done
```

0 stands for the starting index and 2 is the last index not included.

One can append to an array:

```
fruits+=("pineapple")
```

One can remove an element from an array:

```
unset fruits[0]
```

Now the list looks like this:

```
echo "${fruits[@]}"
```

This will output: banana orange pineapple

Finally, we can concatenate arrays like this:

```
fruits=("${fruits[@]:0:2} "${fruits[@]:3}")
```

4.2.2 Associative arrays

Associative arrays are the equivalent of python dictionary.

```
declare -A fruits_price
fruits_price["apple"]=10
fruits_price["banana"]=5
fruits_price["orange"]=8
```

or directly:

```
declare -A fruits_price=(
```

```
  [apple]=10
  [banana]=5
  [orange]=8
)
```

Accessing keys and values:

```
for key in "${!fruits_price[@]}"; do
  value="${fruits_price[$key]}"
  echo "$key: $value"
done
```

To remove a key value pair:

```
unset fruits_price["apple"]
```

Using " " around the key makes the key always valid.

5 Control flow statements

5.1 While loops

```
while condition; do
    # Commands to execute
done
```

Example:

```
#!/usr/bash
emp_num=1
while [ $emp_num -le 1000 ];
do
    cat "$emp_num-dailySales.txt"|grep'Sales_total'|sed 's/.* ://' > "$emp_num-agg.txt"
done
```

5.2 Until loops

```
until condition; do
    # Commands to execute
done
```

5.3 For loops

Example 1:

```
for file in robs_files/*.py
do
    if grep -q 'RandomForestClassifier' $file ; then%  # Move file to folder
        mv $file to_keep/
    fi
done
```

Example 2:

```
for file in output_dir/*results*
```

5.4 Break and continue

As in python, these commands interrupt a loop or skip the rest of the loop.

```
#!/bin/bash
for i in {1..5}; do
    if [ $i -eq 3 ]; then
        echo "Skipping iteration $i"
        continue # Skip the rest of the loop when i is 3
    fi
    echo "Iteration $i"
done
```

Example 3:

```
for x in {1..3}; do
    echo "Number: $x"
done
```

Example 4:

```
array=(1 2 3)
for x in "${array[@]}"; do
    echo "Number: $x"
done
```

Example 5:

```
for (( x=1; x<=3; x++ )); do
    echo "Number: $x"
done
```

5.5 Case statements

General synthax:

```
case variable in
    pattern1)
        # Commands to execute if variable matches pattern1
        ;;
    pattern2)
        # Commands to execute if variable matches pattern2
        ;;
    *)
        # Commands to execute if no patterns match (default case)
        ;;
esac
```

Example:

```
case $(cat $1) in
*sydney*)
mv $1 sydney/ ;;
*melbourne*|*brisbane*)
rm $1 ;;
*canberra*)
mv $1 "IMPORTANT_$1" ;;
*)
echo "No cities found" ;;
esac
```

5.6 If

```
if ; then
    #command
else
    #command
fi
```

Example:

```
if [ "$number" -gt 10 ]; then
    echo "The number is greater than 10."
elif [ "$number" -eq 10 ]; then
    echo "The number is equal to 10."
else
    echo "The number is less than 10."
fi
```

6 Functions

In bash functions have the particularity that, by default, they do not encapsulate the code, ie all the variables are global by default (to make a variable local use the command 'local' before the variable instantiation). Also, there is a return command, but it is not used to send back arguments.

The general synthax is like this:

```
function does_something () {
#code
}
```

How do we pass arguments ?

```
function add_one () {
  local first_par=$1
  (( first_par++ ))
  echo "$first_par"
}
```

As we see, using $1, we recover the first argument in the function. We define first_par as a local variable and then we increment it of one unit. If we were going to use this function we simply would call it like this:

```
> add_one 3
```

To output the result we used the echo command, so that the stdout of the function might be used as the stdin of another function in a pipe.

```
function generate_numbers() {
  for i in {1..5}; do
    echo "$i"
  done
}
function sum_numbers() {
  total=0
  while read number; do
    ((total += number))
  done
  echo "Total: $total"
}

> generate_numbers | sum_numbers
```

Another way to output values in a function is by modifying a global variable:

```
total=0
function sum_to_total() {
  ((total += $1))
}
```

This of course has to be used carefully. Finally the return command, which is optional, is used to mark success (0) or failure (1 to 255).

Last example of function: conversion from nautical miles to km/h

```
function convert_nautical_miles_to_kmh() {
  echo "scale=4; ($1 * 1.852)" | bc
}
```

Using the echo command followed by a string "...", we input to the basic calculator bc a command. There are two aspects, the scale, which are the precision (4 decimals) and the computation.

7 Scripts

Scripts in bash are files with the extension .sh. They have a line called the Shebang line.

```
#!/bin/bash

touch hello.txt

echo "Hello, world!" > hello.txt

echo "File created successfully!"
```

Create this file, name it script.sh, and save it.

Every bash file can be run calling the shell interpreter:

```
> bash create_file.sh
```

Make a script executable means that you can run it directly as a command without explicitly calling the shell interpreter.

To make a script executable change the permissions:

```
chmod +x create_file.sh
```

And run it using:

```
> ./create_file.sh
```

7.1 Virtual environments and PATH variable

When I call the global variable called PATH, it gives me the list of folders that are available system-wide and that can be accessed by all users and processes on the system.

```
> echo $PATH
/home/jacques/Programmes/Anaconda3/bin:/home/jacques/anaconda3/bin:/home/jacques/Programmes/anaconda3/bin:/home/jac
```

When we activate a virtual environment, it modifies the PATH variable. This allows to use the executables and libraries specific to the virtual environment, instead of the global ones.

8 Debugging a bash program

There are several tools to debug a bash program, let's list some:

- In the execution of the program, print the value of the variables. Here we print the variable var:

```
echo "Value of variable: $var"
```

- After a command, we can add the following code:

```
command
if [ $? -ne 0 ]; then
    echo "Command failed"
fi
```

A non nil value means that there is an error.

- At the beginning of a script add set -x, this will print each command and its arguments as they are executed.

```
#!/bin/bash
set -x # Enable debugging

# Your script commands here

set +x
```

-Check synthax (option -n):

```
> bash -n script.sh
```

-Run in debug mode (-x option):

```
> bash -x script.sh
```

9 Help

To get help one can simply use man. For beginners, the level of detail of the documentation is too high, and there are no examples.

```
> man ls
```

I advise therefore to use a large language models such as chatgpt or gemini to get examples and check if the commands are valid.

10 Time based scheduling

10.1 Cron

Cron is a time-based job scheduler. To install it:

```
> sudo apt install cron
> sudo apt install systemctl
```

Start and enable it:

```
> sudo systemctl start cron
> sudo systemctl enable cron
```

Create a Cron job:

```
> sudo crontab -e
```

In the crontab file add lines like this the following ones:

```
0 2 * * * python /path/to/your/script/data_processing.py

46 14 * * * bash /home/jacques/Bureau/cron_test/myscript.sh

echo /path/to/myscript.sh | at 13:55
```

46 14 * * * stands for 46 minutes (0-59), 2 pm (0-23), * any day of the month (1-31), * any month (1-12), * day of Week (0-7, where both 0 and 7 represent Sunday).

In order to check that cron works, you can make a simple script myscript.sh like:

```
#!/bin/bash

now=$(date +"%Y-%m-%d %H:%M:%S")

echo "$now - Cron is working!" >> /home/jacques/Bureau/cron_test/cron_log.txt
```

To stop Cron:

```
> sudo systemctl stop cron
```

To list all the Cron tasks:

```
> crontab -l
```

10.1.1 More advanced scheduling synthax

```
15,30 * * * * bash script.sh
```

This synthax means that the script will run at 0:15 and 0:30, 1:15 and 1:30, 2:15 ...

```
*/30 * * * * bash script.sh
```

This synthax means that the script will every 15 minutes.

10.1.2 Event based scheduling

One can also use cron for event-based scheduling: create a file to signal an event, and then in the crontab add the following line:

```
* * * * * if [ -f "/path/to/event_file" ]; then python
    /path/to/your/script/event_triggered_script.py; rm "/path/to/event_file"; fi
```

11 Mounting a server on each reboot

Let's imagine that you want to mount a server each time you reboot your PC, we want a bash script that does that for us, this script will be called $mount\_server.sh$.

We will need sshfs (secure shell file system):

```
sudo apt install sshfs
```

```
#!/bin/bash

mount_point="/mnt/isaac_server"
server_address= "10.1.100.28" # replace

username="jbourg"  # replace
password="1234"    # replace

sshfs -o IdentityFile=/path/to/your/ssh/key username@server_address:$mount_point

if [ $? -eq 0 ]; then
  echo "Server mounted successfully!"
else
  echo "Failed to mount server. Check your credentials and network connection."
fi
```

Reference this code in rc.local,

```
> sudo nano /etc/rc.local
```

by adding the following line at the end:

```
 /path/to/your/mount_server.sh
```

Then restart the system.

12 The fstab file

The fstab file ("file systems table") is a configuration file in Unix operating systems that defines how devices, and remote filesystems are mounted into the filesystem. This tab is located at /etc/fstab.

Each line follows the following structure <file system> <mount point> <type> <options> <dump> <pass>

If you want to make a modification, it is a good practice to make a copy if it before:

```
> sudo cp /etc/fstab /etc/fstab.bak
```

Then open the file and edit it without changing its permissions:

```
> sudo nano /etc/fstab
```

Before rebooting, do:

```
> sudo mount -a
```

and verify that there are no errors, this could prevent from rebooting properly.

12.1 Adding a NAS in the fstab

First create a folder in the mount folder:

```
sudo mkdir -p /mnt/nas
```

Then if you have a NAS whose network path is //192.168.1.100/share add to your fstab

```
//192.168.1.100/share /mnt/nas cifs
    username=your_username,password=your_password,uid=1000,gid=1000,iocharset=utf8 0 0
```