Bash Script

Some summaries and notes

Eduardo Cunha

Junho 2025

Introduction

This document aims to synthesize the key concepts learned about Bash scripting. Its primary purpose is to clarify, summarize, and reinforce the knowledge acquired, with a particular emphasis on practical applications. The structure of this paper follows the topics as they arise, and not in a strict order.

Fundamentals

Not every Bash script must start with a shebang, but including one is best practice because it tells the system which interpreter to use when executing the script. The most common shebang for Bash scripts is `#!/bin/bash` For scripts written in other languages, such as Python, you would use a different shebang, like `#!/usr/bin/python3` or `#!/usr/bin/env python3`, to specify the appropriate interpreter.

To make a script executable, you need to change its permissions using `chmod +x script.sh`. After that, you can run the script directly.

Variables

Variables are usually defined using uppercase letters, followed by an equal sign (=) with no spaces around it. For example: `NAME="EDUARDO"`

To access a variable, use the dollar sign ($) before the variable name. For example: `echo "Hello, $NAME"`

User Input

To get input from users, the read command is used.

```
read -p "Enter your name: " USERNAME
echo "Welcome, $USERNAME!"
```

Script Arguments

Arguments in Bash scripts work similarly to those in C executables.

- $0 is the name of the script.
- $1, $2, $3, …, represent the first, second, third, and subsequent input arguments.
- $# gives the total number of arguments passed to the script.

Conditional Statements

Example of an simple if statement:

```
if [ "$USER" = "admin" ]; then
  echo "Access granted"
fi
```

Example of an if/else statement:

```
if [ "$1" -gt 100 ]; then
  echo "High"
else
  echo "Low"
fi
```

Comparison Operators

1. For numbers, Bash uses the following operators (placed between the values being compared):

-eq : equal to

-ne : not equal to

-gt : greater than

-lt : less than

-ge : greater than or equal to

-le : less than or equal to

2. For strings, use:

= : equal to

!= : not equal to

-z : string is empty

-n : string is not empty

3. For files, these operators are commonly used (placed behind the filename):

-f : file exists and is a regular file

-d : file exists and is a directory

-e : file or directory exists

-r : file is readable

-w : file is writable

-x : file is executable

Exit Status

Every command returns an exit code ($?):

0 = success

>0 = error/failure

Quoting: " vs '

"Double quotes" allow variable expansion.

'Single quotes' treat contents literally.

```
echo "User: $USER"     # Expands
echo 'User: $USER'     # Literal
```

Comments

Use # to comment

Loops & Control Flow in Bash

"for" Loops

Bash supports two main types of for loops:
1. For-each style (iterating over a list):

```
for file in *.log; do
  echo "Processing $file"
done
```

2. C-style for loop (using a counter):

```
for (( i=0; i<5; i++ )); do
  echo "Counter: $i"
done
```

"while" Loops

A while loop executes as long as the condition is true:

```
count=0
while [ $count -lt 3 ]; do
  echo "Count is $count"
  count=$((count + 1))
done
```

“until” Loops

An until loop is similar to a while loop, but it runs until the condition becomes true (i.e., it continues while the condition is false):

```
n=0
until [ $n -ge 5 ]; do
  echo "n = $n"
  n=$((n + 1))
done
```

Loop Control

- “break” exits the loop immediately.
- “continue” skips to the next iteration of the loop.

Logical Operators

- && : AND
- || : OR
- ! : NOT

example of usage:

```
if [ -f "$file" ] && [ -r "$file" ]; then
  echo "$file exists and is readable"
fi
```

[[vs [– Safer Conditional Syntax

In Bash, it’s generally better to use [[…]] for conditionals instead of […]. The [[…]] syntax is more powerful and safer because:

- It is less prone to syntax errors (e.g., with spaces or special characters).
- It supports advanced features like regular expression matching.
- It does not perform word splitting or pathname expansion.

Example:

```
if [[ "$input" =~ ^[0-9]+$ ]]; then
  echo "Numeric input"
fi
```

Functions

Functions help you organize, reuse, and modularize your code

Defining a Function

Basic syntax:

```
my_function() {
  echo "Hello from the function"
}
```

or

```
function my_function {
  echo "Alternative syntax"
}
```

Call it like: “my_function”

Arguments

Inside a function, arguments are accessed like in scripts:

```
greet_user() {
  echo "Hello, $1!"
}

greet_user "Alice"
```

Returning Values

There are two common ways to return values from Bash functions:

1. Return Status (Exit Codes)

Use the “return” statement to provide a numeric exit code (between 0 and 255). This is typically used to indicate success or failure, not to return data.

2. Return Text (Using echo)

Use “echo” inside the function to output a value, and capture it with command substitution. This is the standard way to return strings or other data from a function.

Scope of Variables

By default, variables are global inside Bash scripts, even in functions.

Use “local” to avoid polluting global scope.

Using return vs exit

- return is used inside functions.
- exit to stop the entire script.

Function Libraries

You can source functions from other files by using the “source” command.

Arrays & String Manipulation

Declaring Arrays

```
fruits=("apple" "banana" "cherry")
```

You can also assign individual elements:

```
fruits[3]="orange"
```

Accessing Array Elements

```
${fruits[2]}
```

Indexing starts at 0.

All Elements and Length

```
echo "${fruits[@]}"      # All elements
echo "${#fruits[@]}"     # Number of elements
```

Array Slicing

```
echo "${fruits[@]:1:2}"   # array[@]:start:length
```

Substring Extraction

```
str="security"
echo "${str:0:3}"    # sec
echo "${str:3}"      # urity
```

String Length

```
echo "${#str}"   # 8
```

String Replacement

```
text="fail2ban is active"
echo "${text/active/inactive}"      # One occurrence
echo "${text//a/_}"                 # All 'a' replaced by '_'
```

Input/Output (I/O) & Redirection in Bash

Standard I/O Streams

In Unix/Linux systems, every program works with three standard data streams, each identified by a number:

Name	Number	Purpose
stdin	0	input
stdout	1	output
stderr	2	error

Output Redirection

- “>” : Overwrite output to file
- “>>” : Append output to file

Input Redirection

- “<” : “command < input.txt”

Read Files

You can read from files line by line with “while” and “read”:

```
while read -r line; do
  echo "Line: $line"
done < /etc/passwd
```

This example reads the “/etc/passwd” file line by line.

Common Tools

This chapter introduces essential command-line tools often used inside Bash scripts for log parsing, monitoring, text manipulation, etc…

grep

Search for lines matching a pattern (supports regex).

```
grep "ERROR" /var/log/syslog
grep -i "failed" auth.log        # case-insensitive
```

awk

Process structured text (like logs, CSV, or system files).

```
awk '{ print $1 }' /var/log/nginx/access.log     # Example: Extract IPs from a log
```

In awk, field numbering starts at 1, not zero.

sed

Stream Editing (Find & Replace)

```
sed 's/old/new/' file.txt             # replace first match per line
sed 's/old/new/g' file.txt            # replace all matches
```

sort, uniq

Organizing Output

```
sort users.txt | uniq        # remove duplicates
sort -nr file.txt            # sort numerically (descending)
```

"uniq -c" only works on a sorted input.

find

Search Files with Conditions

```
find /var/log -name "*.log"
find /home -type f -size +100M
```

wc

Count Lines, Words, Characters

```
wc -l /var/log/syslog     # count lines
wc -w myfile.txt          # count words
```

head, tail

View File Start/End

```
head -n 10 file.txt           # first 10 lines
tail -n 20 /var/log/syslog    # last 20 lines
```

stat, du

File Information & Size

```
stat myfile.txt            # metadata (modified time, size, permissions)
du -sh /var/log            # human-readable size
```

chmod, chown, ls, file

chmod +x script.sh — make executable

chown root:root file — change ownership

ls -l — long list with permissions

file /bin/bash — detect file type

ssh, scp

Remote Access and File Transfer

```
ssh user@host              # remote shell
scp file.txt user@host:/tmp # copy file remotely
```

ps, top, htop, kill

ps aux — shows all running processes on the system.

top — displays active processes and system resource usage in real time.

htop — an interactive and colorful version of top.

kill PID — sends a signal to terminate (by default, SIGTERM) the process with the given PID.

cat

Concatenates and displays the entire contents of one or more files.

nano, vim

Text editors for quick in-terminal edits.

tr

Translates, deletes, or squeezes characters from input streams.

```
tr ',' '\t'            # Convert commas to tabs
```

echo

Displays a line of text or variables to standard output.

```
echo "Hello, world"         # Print plain text
echo $HOME                  # Print value of an environment variable
echo -e "Line1\nLine2"      # Enable interpretation of backslash escapes
echo $(date)                # Print output of a command
```

Pipes ("|")

Pipes connect the standard output (stdout) of one command directly to the standard input (stdin) of another, allowing you to chain commands and process data in stages without creating intermediate files.

Challenge

Bash Challenge: Filter and Display Specific Log Lines

You have a log file called app.log. Each line in this log file follows the format:

`[2025-07-01 12:34:56] LEVEL: Message`

Your task is to write a Bash script that takes two arguments: the name of the log file (e.g., app.log) and the log level to filter (e.g., ERROR, INFO, WARNING). The script must include a function called filter_lines that receives the filename and the log level as arguments, and prints all lines from the file that exactly match the specified level (case-insensitive). The script should also display how many lines were found, and if none were found, it should print an appropriate message.

Answer

```bash
#!/bin/bash

# filename - app.log
# line example - [2025-07-01 12:34:56] LEVEL: message

FILENAME="$1"
LEVEL="$2"

line_filter() {
    local file="$1"
    local level="$2"

    local lines
    lines=$(grep -i "\] $level:" "$file")

    local count
    count=$(echo "$lines" | wc -l)

    if [ "$count" -eq 0 ]; then
        echo "None"
    else
        echo "$lines"
        echo "Total: $count lines."
    fi
}

line_filter "$FILENAME" "$LEVEL"
```