**Shell Learnings:**

a) $? – Exit status of the last command, if it is zero, last command was successful.

b) $# -- total no. of args passed to the shell script.

c) Storing the output of a command to a variables. (var=$(ls -lrt);echo $) or (var=’ls -lrt’)

1) How to make variable readonly or **unchangeable:**

$a=10

$readonly a

2) **How variables can be wiped out?**

Variables can be wiped out or erased using the **unset** command.

**Example:**  
$ a =20  
$ unset a

Upon using the above command the variable ‘**a**’ and its value **20**get erased from shell’s memory.

3) **Generally, each block in UNIX is how many bytes?**

Each block in UNIX is 1024 bytes.

4) **By default, a new file and a new directory that is being created will have how many links?**

New file contains one link. And a new directory contains two links

5) What are the different blocks of a file system? Explain in brief:

**Answer:** **Given below are the main 4 different blocks available on a file system.**

| **File System** |  |
| --- | --- |
| **Block No.** | Name of the Block |
| **1st Block** | Boot Block |
| **2nd Block** | Super Block |
| **3rd Block** | Inode Table |
| **4th Block** | Data Block |

* **Super Block**: This block mainly tells about a state of the file system like how big it is, maximum how many files can be accommodated, etc.
* **Boot Block**: This represents the beginning of a file system. It contains the bootstrap loader program, which gets executed when we boot the host machine.
* **Inode Table**: As we know all the entities in a UNIX are treated as files. So, the information related to these files is stored in an Inode table.
* **Data Block**: This block contains the actual file contents.

**6) )** What is the alternative command available to echo and what does it do?

**Answer:** **tput**is an alternative command to **echo**.

Using this, we can control the way in which the output is displayed on the screen.

7) **Q #25) By default, a new file and a new directory that is being created will have how many links?**

**Answer:** New file contains one link. And a new directory contains two links.

8) **What is the alternative command available to echo and what does it do?**

**Answer:** **tput**is an alternative command to **echo**.

9) **What are control instructions and how many types of control instructions are available in a shell? Explain in brief.**

**Answer:** Control Instructions are the ones, which enable us to specify the order in which the various instructions in a program/script are to be executed by the computer. Basically, they determine a flow of control in a program.

**There are 4 types of control instructions that are available in a shell.**

* **Sequence Control Instruction**: This ensures that the instructions are executed in the same order in which they appear in the program.
* **Selection or Decision Control Instruction**: It allows the computer to take the decision as to which instruction is to be executed next.
* **Repetition or Loop Control Instruction**: It helps a computer to execute a group of statements repeatedly.
* **Case-Control Instruction**: This is used when we need to select from several alternatives.

10) **36) What is IFS?**

**Answer:** IFS stands for Internal Field Separator. And it is one of the system variables. By default, its value is space, tab, and a new line. It signifies that in a line where one field or word ends and another begins.

11) **#41) Which command needs to be used to know how long the system has been running?**

**Answer:** **uptime**command needs to be used to know how long the system has been running.

**Example:** $ uptime

On entering the above command at shell prompt i.e. $ uptime, the output should look like this.

9:21am  up 86 day(s), 11:46,  3 users,  load average: 2.24, 2.18, 2.16

12) **2) How to find the current shell which you are using?**

**Answer:** We can find the current shell that we are using with echo $SHELL.

**Example:** $ echo $SHELL

**Execution over Shell Interpreter/Editor**

Current Shell Script

**Output**:

13) **43) How to find all the available shells in your system?**

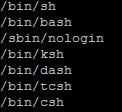
**Answer:** We can find all the available shells in our system with $ cat /etc/shells.

**Example:** $ cat /etc/shells

**Execution over Shell Interpreter/Editor**

Available Shell Script

**Output**:



14) Switch the shell: “chsh” command

15) **#45) How many fields are present in a crontab file and what does each field specify?**

**Answer:** The **crontab** file has six fields. The first five fields tell **cron** when to execute the command: **minute(0-59), hour(0-23), day(1-31), month(1-12), and day of the week(0-6, Sunday = 0).**

And the sixth field contains the command to be executed.

**Q #46) What are the two files of crontab command?**

**Answer:** **Two files of crontab command are**:

* **cron.allow** – It decides which users need to be permitted from using crontab command.
* **cron.deny** – It decides which users need to be prevented from using crontab command.

**7) What command needs to be used to take the backup?**

* **Answer:** **tar** is the command which needs to be used to take the backup. It stands for tape archive. The **tar**command is mainly used to save and restore files to and from an archive medium like tape.
* **#50) How to find out the total disk space used by a specific user, say for example username is John?**
* **Answer:** The total disk space used by John can be found out as:
* **du –s/home/John**

**#52) What is the command to be used to display the shell’s environment variables?**

**Answer:** Command to be used to display the shell’s environment variables is **env** or **printenv**.

**#53) How to debug the problems encountered in shell script/program?**

**Answer:** Though it depends on the type of problem encountered. Given below are some common methods used to debug the problems in the script.

* Debug statements can be inserted in the shell script to output/display the information which helps to identify the problem.
* Using “set -x” we can enable debugging in the script.

**54) How to know the variable length?**

**Answer:** Variable length can be checked by **$ {#variable}**

**#57) How can the contents of a file inside jar be read without extracting in a shell script?**

**Answer:**The contents of the file inside a jar can be read without extracting in a shell script as shown below.  
tar –tvf <File Name>.tar

**#58) What is the difference between diff and cmp commands?**

**Answer:** **diff** – Basically, it tells about the changes which need to be made to make files identical.

**cmp** – Basically it compares two files byte by byte and displays the very first mismatch.

**#59) Explain in brief about sed command with an example.**

**Answer:** **sed** stands for **stream editor**. And it is used for editing a file without using an editor. It is used to edit a given stream i.e. a file or input from a pipeline.

**Syntax**: sed options file

**Example:**

**Execution over Shell Interpreter/Editor**

Sed command

Here ‘**s’** command present in **sed** will replace string **Hello** with **Hi**.

**Output**:

**#60) Explain in brief about awk command with an example.**

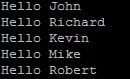
**Answer:** **awk** is a data manipulation utility or command. Hence, it is used for data manipulation.

**Syntax**: awk options File Name

**Example:**

**Script/Code**

Awk Command



awk utility/command assigns variables like this.  
$0 -> For whole line (e.g. Hello John)  
$1 -> For the first field i.e. Hello  
$2 -> For the second field

**What is the significance of the Shebang line in Shell Scripting?**

The Shebang line is present at the top of the script,e.g. *#!/bin/sh*. It simply provides information regarding the location where the engine is placed. The engine is the one that executes the script.

For loop syntax:

*for var in word1 word2 ... wordN*

*do*

*Statement(s) to be executed for every word.*

*Done*

**While Loop:**

*while command*

*do*

*Statement(s) to be executed if command is true*

*done*

**Until Loop:**

until command

do

Statement(s) to be executed until command is true

done

**Q22. How would you compare the strings in a Shell Script?**

The **test command** is used to compare the text strings. The test command compares text strings by comparing each character in each string.

**Q28. What are the default permissions of a file when it is created?**

On Linux and other Unix-like operating systems, new files are created with a default set of permissions. The **umask** or user mask command is used to determine the default permissions for newly created files. It is a 4-digit Octal number which is set and expressed using symbolic values. The default permission of a file when it is created is **664** i.e. **rw-rw-r-**. The table for file permissions is given below;

|  |  |  |
| --- | --- | --- |
| **0** | **0** | **No permissions** |
| **1** | **1** | **execute** |
| **2** | **2** | **write** |
| **3** | **1+2** | **execute + write** |
| **4** | **4** | **read** |
| **5** | **1+4** | **execute + read** |
| **6** | **2+4** | **write + read** |
| **7** | **1+2+4** | **execute + write + read** |

**Q34. How to get part of string variable with echo command only?**

*#!/bin/sh*  
*echo ${variable:x:y}*  
*#x - start position*  
*#y - length*

*variable="My name is Upasana, and I work at Edureka."*  
*echo ${variable:11:7} # will display Upasana*

**Q38. How to print all array elements and their respective indexes?**

*!/bin/sh*  
*array=("This" "is" "Shell" "Scripting")*  
*echo ${array[@]}*  
*echo ${!array[@]}*

**Q39. How to print the first array element?**

*#!/bin/sh*  
*array=("This" "is" "Shell" "Scripting" )*  
*echo ${array[0]}*

**Q58. How to check if a directory exists?**

*#!/bin/sh*

*if [ -d $mydir ]*  
*then*  
*echo "Directory exists"*  
*fi*

**Functions in shell scripting:**

A function may return a value in one of five typical ways:

* Change the state of a variable or variables
* Use the exit command to end the shell script
* Update the contents of a file in the filesystem
* Use the return command to end the function, and return the supplied value to the calling section of the shell script
* echo output to stdout, which will be caught by the caller just as c=`expr $a + $b` is caught

This is rather like C, in that exit stops the program, and return returns control to the caller. The difference is that a shell function cannot change its parameters, though it can change global variables.

a) calling a function:

./test.sh

myfunc(){

echo $# // printing the no.of args as

echo $1

echo $2

echo $3

}

echo “stating the script”

myfunc 1 2 shankar

b) **Array in Shell Scripting**  
An array is a systematic arrangement of the same type of data. But in Shell script Array is a variable which contains multiple values may be of same type or different type since by default in shell script everything is treated as a string. An array is zero-based ie indexing start with 0.

**How to Declare Array in Shell Scripting?**  
We can declare an array in a shell script in different ways.

1. Indirect Declaration  
In Indirect declaration, We assigned a value in a particular index of Array Variable. No need to first declare.

ARRAYNAME[INDEXNR]=value

2. Explicit Declaration  
In Explicit Declaration, First We declare array then assigned the values.

declare -a ARRAYNAME

3. Compound Assignment  
In Compount Assignment, We declare array with a bunch of values. We can add other values later too.

ARRAYNAME=(value1 value2 .... valueN)

or  
[indexnumber=]string

ARRAYNAME=([1]=10 [2]=20 [3]=30)

**To Print Array Value in Shell Script?**

To Print All elements  
[@] & [\*] means All elements of Array.

echo ${ARRAYNAME[\*]}

**Printing the new line in shell:**

echo -e "Hello\nworld"

echo -e 'Hello\nworld'

echo Hello$'\n'world

echo Hello ; echo world

printf "hello\nworld\n"

printf has more consistent behavior than echo. The behavior of echo varies greatly between different versions.

**AWK Tutorial:**

## *Program Structure*

*Let us now understand the program structure of AWK.*

### *BEGIN block*

*The syntax of the BEGIN block is as follows −*

*Syntax*

*BEGIN {awk-commands}*

*The BEGIN block gets executed at program start-up. It executes only once. This is good place to initialize variables. BEGIN is an AWK keyword and hence it must be in upper-case. Please note that this block is optional.*

### *Body Block*

*The syntax of the body block is as follows −*

*Syntax*

*/pattern/ {awk-commands}*

*The body block applies AWK commands on every input line. By default, AWK executes commands on every line. We can restrict this by providing patterns. Note that there are no keywords for the Body block.*

### *END Block*

*The syntax of the END block is as follows −*

*Syntax*

*END {awk-commands}*

*The END block executes at the end of the program. END is an AWK keyword and hence it must be in upper-case. Please note that this block is optional.*

*Let us create a file marks.txt which contains the serial number, name of the student, subject name, and number of marks obtained.*

*1) Amit Physics 80*

*2) Rahul Maths 90*

*3) Shyam Biology 87*

*4) Kedar English 85*

*5) Hari History 89*

*Let us now display the file contents with header by using AWK script.*

*Example*

*[jerry]$ awk 'BEGIN{printf "Sr No\tName\tSub\tMarks\n"} {print}' marks.txt*

*When this code is executed, it produces the following result −*

*Output*

*Sr No Name Sub Marks*

*1) Amit Physics 80*

*2) Rahul Maths 90*

*3) Shyam Biology 87*

*4) Kedar English 85*

*5) Hari History 89*

a) If a file is there with below content: (myfile.txt)

1) Amit Physics 80

2) Rahul Maths 90

3) Shyam Biology 87

4) Kedar English 85

5) Hari History 89

b)Command to print complete file is “awk ‘{print}’ myfile.txt

c) We can provide awk commands in a script file such as “commands.awk”

file has content as {print}

Execute the file using the command : “awk -f command.awk marks.txt”

d) standard options:

AWK supports the following standard options which can be provided from the command line.

### The -v option

This option assigns a value to a variable. It allows assignment before the program execution. The following example describes the usage of the -v option.

awk -v name=Jerry 'BEGIN{printf "Name = %s\n", name}'

### e) The --dump-variables[=file] option

It prints a sorted list of global variables and their final values to file. The default file is **awkvars.out**.

[jerry]$ awk --dump-variables ''

[jerry]$ cat awkvars.out

f) awk environment variables:

*Usage: awk [POSIX or GNU style options] -f progfile [--] file ...*

*Usage: awk [POSIX or GNU style options] [--] 'program' file ...*

*POSIX options : GNU long options: (standard)*

*-f progfile --file=progfile*

*-F fs --field-separator=fs*

*-v var=val --assign=var=val*

*Short options : GNU long options: (extensions)*

*-b --characters-as-bytes*

*-c --traditional*

*-C --copyright*

*-d[file] --dump-variables[=file]*

*-e 'program-text' --source='program-text'*

*-E file --exec=file*

*-g --gen-pot*

*-h --help*

*-L [fatal] --lint[=fatal]*

*-n --non-decimal-data*

*-N --use-lc-numeric*

*-O --optimize*

*-p[file] --profile[=file]*

*-P --posix*

*-r --re-interval*

*-S --sandbox*

*-t --lint-old*

*-V --version*

g) Printing You can instruct AWK to print only certain columns from the input field. The following example demonstrates this , below prints 3rd and 4th column with tab as separator from the marks.txt file

awk '{print $3 "\t" $4}' marks.txt

h) Printing all lines: awk '/a/ {print $0}' marks.txt

i) Counting and Printing Matched Pattern

Let us see an example where you can count and print the number of lines for which a pattern match succeeded.

Example

[jerry]$ awk '/a/{++cnt} END {print "Count = ", cnt}' marks.txt

In this example, we increment the value of counter when a pattern match succeeds and we print this value in the END block. Note that unlike other programming languages, there is no need to declare a variable before using it.

j) Printing Lines with More than 18 Characters

Let us print only those lines that contain more than 18 characters.

awk 'length($0) > 18' marks.txt

AWK provides a built-in **length** function that returns the length of the string. **$0** variable stores the entire line and in the absence of a body block, default action is taken, i.e., the print action. Hence, if a line has more than 18 characters, then the comparison results true and the line gets printed.

h) AWK provides several built-in variables. They play an important role while writing AWK scripts. This chapter demonstrates the usage of built-in variables.

## Standard AWK variables

The standard AWK variables are discussed below.

### ARGC

It implies the number of arguments provided at the command line.

Example

[jerry]$ awk 'BEGIN {print "Arguments =", ARGC}' One Two Three Four

On executing this code, you get the following result −

Output

Arguments = 5

But why AWK shows 5 when you passed only 4 arguments? Just check the following example to clear your doubt.

### ARGV

It is an array that stores the command-line arguments. The array's valid index ranges from 0 to ARGC-1.

Example

[jerry]$ awk 'BEGIN {

for (i = 0; i < ARGC - 1; ++i) {

printf "ARGV[%d] = %s\n", i, ARGV[i]

}

}' one two three four

On executing this code, you get the following result −

Output

ARGV[0] = awk

ARGV[1] = one

ARGV[2] = two

ARGV[3] = three

### CONVFMT

It represents the conversion format for numbers. Its default value is %.6g.

Example

[jerry]$ awk 'BEGIN { print "Conversion Format =", CONVFMT }'

On executing this code, you get the following result −

Output

Conversion Format = %.6g

### ENVIRON

It is an associative array of environment variables.

Example

[jerry]$ awk 'BEGIN { print ENVIRON["USER"] }'

On executing this code, you get the following result −

Output

jerry

To find names of other environment variables, use env command.

### FILENAME

It represents the current file name.

Example

[jerry]$ awk 'END {print FILENAME}' marks.txt

On executing this code, you get the following result −

Output

marks.txt

Please note that FILENAME is undefined in the BEGIN block.

### FS

It represents the (input) field separator and its default value is space. You can also change this by using -F command line option.

Example

[jerry]$ awk 'BEGIN {print "FS = " FS}' | cat -vte

On executing this code, you get the following result −

Output

FS = $

### NF

It represents the number of fields in the current record. For instance, the following example prints only those lines that contain more than two fields.

Example

[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NF > 2'

On executing this code, you get the following result −

Output

One Two Three

One Two Three Four

### NR

It represents the number of the current record. For instance, the following example prints the record if the current record number is less than three.

Example

[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NR < 3'

On executing this code, you get the following result −

Output

One Two

One Two Three

### FNR

It is similar to NR, but relative to the current file. It is useful when AWK is operating on multiple files. Value of FNR resets with new file.

### OFMT

It represents the output format number and its default value is %.6g.

Example

[jerry]$ awk 'BEGIN {print "OFMT = " OFMT}'

On executing this code, you get the following result −

Output

OFMT = %.6g

### OFS

It represents the output field separator and its default value is space.

Example

[jerry]$ awk 'BEGIN {print "OFS = " OFS}' | cat -vte

On executing this code, you get the following result −

Output

OFS = $

### ORS

It represents the output record separator and its default value is newline.

Example

[jerry]$ awk 'BEGIN {print "ORS = " ORS}' | cat -vte

On executing the above code, you get the following result −

Output

ORS = $

$

### RLENGTH

It represents the length of the string matched by match function. AWK's match function searches for a given string in the input-string.

Example

[jerry]$ awk 'BEGIN { if (match("One Two Three", "re")) { print RLENGTH } }'

On executing this code, you get the following result −

Output

2

### RS

It represents (input) record separator and its default value is newline.

Example

[jerry]$ awk 'BEGIN {print "RS = " RS}' | cat -vte

On executing this code, you get the following result −

Output

RS = $

$

### RSTART

It represents the first position in the string matched by match function.

Example

[jerry]$ awk 'BEGIN { if (match("One Two Three", "Thre")) { print RSTART } }'

On executing this code, you get the following result −

Output

9

### SUBSEP

It represents the separator character for array subscripts and its default value is \034.

Example

[jerry]$ awk 'BEGIN { print "SUBSEP = " SUBSEP }' | cat -vte

On executing this code, you get the following result −

Output

SUBSEP = ^\$

### $0

It represents the entire input record.

Example

[jerry]$ awk '{print $0}' marks.txt

On executing this code, you get the following result −

Output

1) Amit Physics 80

2) Rahul Maths 90

3) Shyam Biology 87

4) Kedar English 85

5) Hari History 89

### $n

It represents the nth field in the current record where the fields are separated by FS.

Example

[jerry]$ awk '{print $3 "\t" $4}' marks.txt

On executing this code, you get the following result −

Output

Physics 80

Maths 90

Biology 87

English 85

History 89

## GNU AWK Specific Variables

GNU AWK specific variables are as follows −

### ARGIND

It represents the index in ARGV of the current file being processed.

Example

[jerry]$ awk '{

print "ARGIND = ", ARGIND; print "Filename = ", ARGV[ARGIND]

}' junk1 junk2 junk3

On executing this code, you get the following result −

Output

ARGIND = 1

Filename = junk1

ARGIND = 2

Filename = junk2

ARGIND = 3

Filename = junk3

### BINMODE

It is used to specify binary mode for all file I/O on non-POSIX systems. Numeric values of 1, 2, or 3 specify that input files, output files, or all files, respectively, should use binary I/O. String values of r or w specify that input files or output files, respectively, should use binary I/O. String values of rw or wr specify that all files should use binary I/O.

### ERRNO

A string indicates an error when a redirection fails for getline or if close call fails.

Example

[jerry]$ awk 'BEGIN { ret = getline < "junk.txt"; if (ret == -1) print "Error:", ERRNO }'

On executing this code, you get the following result −

Output

Error: No such file or directory

### FIELDWIDTHS

A space separated list of field widths variable is set, GAWK parses the input into fields of fixed width, instead of using the value of the FS variable as the field separator.

### IGNORECASE

When this variable is set, GAWK becomes case-insensitive. The following example demonstrates this −

Example

[jerry]$ awk 'BEGIN{IGNORECASE = 1} /amit/' marks.txt

On executing this code, you get the following result −

Output

1) Amit Physics 80

### LINT

It provides dynamic control of the --lint option from the GAWK program. When this variable is set, GAWK prints lint warnings. When assigned the string value fatal, lint warnings become fatal errors, exactly like --lint=fatal.

Example

[jerry]$ awk 'BEGIN {LINT = 1; a}'

On executing this code, you get the following result −

Output

awk: cmd. line:1: warning: reference to uninitialized variable `a'

awk: cmd. line:1: warning: statement has no effect

### PROCINFO

This is an associative array containing information about the process, such as real and effective UID numbers, process ID number, and so on.

Example

[jerry]$ awk 'BEGIN { print PROCINFO["pid"] }'

On executing this code, you get the following result −

Output

4316

### TEXTDOMAIN

It represents the text domain of the AWK program. It is used to find the localized translations for the program's strings.

Example

[jerry]$ awk 'BEGIN { print TEXTDOMAIN }'

On executing this code, you get the following result −

Output

messages

The above output shows English text due to en\_IN locale

What exit code should I use?

The Linux Documentation Project has a list of [reserved codes](http://www.tldp.org/LDP/abs/html/exitcodes.html) that also offers advice on what code to use for specific scenarios. These are the standard error codes in Linux or UNIX.

* 1 - Catchall for general errors
* 2 - Misuse of shell builtins (according to Bash documentation)
* 126 - Command invoked cannot execute
* 127 - “command not found”
* 128 - Invalid argument to exit
* 128+n - Fatal error signal “n”
* 130 - Script terminated by Control-C
* 255\\* - Exit status out of range

How to suppress exit statuses

Sometimes there may be a requirement to suppress an exit status. It may be that a command is being run within another script and that anything other than a 0 status is undesirable.

In the following example a file is printed to the terminal using [cat](https://shapeshed.com/unix-cat/). This file does not exist so will cause an exit status of 1.

To suppress the error message any output to standard error is sent to /dev/null using 2>/dev/null.

If the cat command fails an OR operation can be used to provide a fallback - cat file.txt || exit 0. In this case an exit code of 0 is returned even if there is an error.

Combining both the suppression of error output and the OR operation the following script returns a status code of 0 with no output even though the file does not exist.

#!/bin/bash

cat 'doesnotexist.txt' 2>/dev/null || exit 0

# How to capture terminal sessions and output with the Linux script command

The Linux script command allows you to create replayable terminal sessions by simply entering commands.

Posted: January 27, 2021 | **by**[Ken Hess (Red Hat)](https://www.redhat.com/sysadmin/users/khess)

Image



Photo by [**Sora Shimazaki**](https://www.pexels.com/@sora-shimazaki?utm_content=attributionCopyText&utm_medium=referral&utm_source=pexels) from [**Pexels**](https://www.pexels.com/photo/crop-cyber-spy-hacking-system-while-typing-on-laptop-5935794/?utm_content=attributionCopyText&utm_medium=referral&utm_source=pexels)

The Linux script command creates a typescript file from your terminal session. This means that if you invoke the script command, you are dropped to a "watched and recorded" terminal session subshell that's saved to an ASCII text file. When created with a timing file, you can replay the session, including output. The purpose of script is that you can easily grab sample output from any command through an interactive session exactly as it's displayed in your terminal. You can use backspace, edit files, create files, and run simple or complex commands.

***[ Readers also liked:***[***Linux Command Basics: 7 commands for process management***](https://www.redhat.com/sysadmin/linux-command-basics-7-commands-process-management)***]***

## **Linux Containers**

* [Free course: Deploying containerized applications](https://www.redhat.com/en/services/training/do080-deploying-containerized-applications-technical-overview?intcmp=701f20000012ngPAAQ)
* [Download now: Red Hat OpenShift Trial](https://www.redhat.com/en/technologies/cloud-computing/openshift/try-it?intcmp=701f20000012ngPAAQ)
* [More on containers](https://www.redhat.com/sysadmin/topics/containers?intcmp=701f20000012ngPAAQ)

The value of the script command is in its capability to capture output during your terminal session for any terminal command without redirects, which don't always work. I was frustrated so many times when attempting to capture output from a command that somehow is going awry until I discovered script. With standard redirect operators, some output can be redirected to a file, while other commands will only show output in stdout or the screen. Most sysadmins use the script command to show output during software installation, when troubleshooting, or for development and programming purposes.

Surprisingly, the script command does not help you create shell scripts.

## **Script options**

As with most commands that I use, I only use a subset of available options for them. The script command has several options that I've never found useful in my own work. The only ones I use are:

* -a for appending new commands and output to a previously-used file.
* -q for removing the initial starting and ending statements when using script.
* --t for saving timing information for playback.

When I use script, I always use --t to create a timing file and -q for quiet mode. I only use -a when I need to append more info into an existing script file, which is rare.

## **Script usage**

The following are two standard examples of the way I use script:

**$** script --t=<logfile> -q <script file>

And, to append to script file:

**$** script --t=<logfile> -q -a <script file>

Where logfile and script file can be names that you choose. When you want to end and save the file, use **Ctrl-D** on your keyboard. You can look at, edit, or remove the script file and the log file at will. They are simple ASCII text files.

Here is an example:

**$** script --t=script\_log -q scriptfile

I ran the ls command, the who command, and then I ended the script with **Ctrl-D**.

**$** ls

blah.txt test1 test2 doc.txt

**$** who

root tty1 2021-01-18 09:31

khess pts/0 2021-01-20 14:42 (192.168.0.5)

khess pts/1 2021-01-20 14:47

**$** exit

When you press **Ctrl-D**, the script exits and displays **exit**.

Use the cat command to display the contents of scriptfile.

**$** ls

blah.txt file\_time scriptfile script.rec shell\_record1 shell\_record3 time\_log

file\_log record.scr script\_log scriptrecord shell\_record2 snap typescript

**$** who

root tty1 2021-01-18 09:31

khess pts/0 2021-01-20 14:42 (192.168.0.5)

khess pts/1 2021-01-20 14:47

**$** exit

Script done on 2021-01-20 14:47:28-06:00

If you want, you can also cat the script\_log file.

**$** cat script\_log

0.088699 31

3.393729 1

0.246070 1

0.540094 2

0.003060 196

0.000195 31

2.136900 1

0.177266 1

0.179336 1

0.540818 2

0.003883 134

0.000210 31

4.676286 6

## **Kubernetes and OpenShift**

* [Free cheatsheet: Kubernetes and Minikube](https://developers.redhat.com/cheat-sheets/kubernetes?intcmp=7013a0000026EKuAAM)
* [Free ebook: Designing Cloud-Native Applications](https://developers.redhat.com/books/kubernetes-patterns?intcmp=7013a0000026EKuAAM)
* [Interactive course: Getting Started with OpenShift](https://developers.redhat.com/courses/openshift/getting-started?intcmp=7013a0000026EKuAAM)
* [Free ebook: Build Applications with Kubernetes and Openshift](https://developers.redhat.com/books/knative-cookbook?intcmp=7013a0000026EKuAAM)

This is the timing log file that behaves similar to a transaction log for your script commands and responses. It is important when you play back the file, which I demonstrate in the follow-up article, [How to replay terminal sessions recorded with the Linux script command](https://redhat.com/sysadmin/playback-scriptreplay).

***[ Learn the basics of using Kubernetes in this***[***free cheat sheet***](https://developers.redhat.com/promotions/kubernetes-cheatsheet?intcmp=701f20000012ngPAAQ)***. ]***

## **Wrap up**

For me, the best application of the script command is for training new Linux users on how to use commands and to show them expected output in real-time, as if they were interacting with the terminal session themselves. For more experienced users, you could create a training session that teaches a new software installation or configuration. Training is the application I think of because of my history with training new sysadmins and writing how-to articles for various venues. And since the output is in ASCII text files, you can change the output for your own needs and audiences.