Shell Scripting

A shell script is a computer program designed to be run by the Unix shell, a command-line interpreter. Use this guide to get a great hold on shell scripting!

Index Of Contents

1. Introduction to scripts
2. Our first script
3. Variables
    1. Syntax
    2. Examples
    3. Valid Variable Names
4. Tests
    1. Syntax
    2. File Operations
    3. String Operations
    4. Arithmetic Operations
5. Making Decisions Based On Conditions
    1. The IF Statement
        - Syntax for IF Statement
        - IF-Else Tree
        - IF-Elif-Else Ladder
    2. Case statements
        - Syntax for case statements
        - When to use case statements
6. Iterative Statements
    1. The For loop
    2. The While loop
    3. Infinite loop
    4. Examples
    5. The continue statement
    6. The break statement
7. Positional Parameters
8. Exit Statuses
    1. Logical Operations
    2. The Semicolon
    3. The Exit Command
9. Functions in shell
    1. Syntax for creating functions
    2. Calling a function from another function
    3. Positional Parameters In Functions
    4. Scope of a variable
    5. Return codes for function
10. Wildcards
    1. What are wildcards

    2. Some Commonly Used Wildcards
    3. Predefined named character classes
    4. Using wildcards in shell scripts
11. Logging
    1. Syslog
    2. Facilities
    3. Severities
    4. The Logger Command
12. Debugging
13. Sample Programs For Revision
14. Download free PDF

Scripts

You might have came across the word 'script' a lot of times, but what is the meaning of a script? So basically, a script is a command line program that contains a series of commands to be executed. These commands are execued by an interpreter. Anything you can put into a command line, you can put in a script. And, scripts are great for automating tasks. If you find yourself repeating some commands frequently, you can, rather you should, create a script for doing it!

Our first script

```bash
#!/bin/bash
echo "My First Script!"
```

To run it:

```
$ chmod 755 script.sh
$ ./script.sh
```

Find the code here

Shebang

A script starts with #! **Path To Bash**

`#` is often called sharp and `!` is called Bang, hence the name sharp bang, but generally people say it **shebang** instead of sharp bang.

Comments

Comments are started by a `#` sign, anything after pound sign on that line is ignored.

Example

```
#!/bin/bash
echo "Hello World!"
# This line won be executed!
```

Basic Examples Of Shell Scripts

- Using csh as interpreter

```
#!/bin/csh
echo "This script uses csh as the interpreter!"
```

- Using ksh as interpreter

```
#!/bin/ksh
echo "This script uses ksh as the interpreter!"
```

- Using zsh as interpreter

```
#!/bin/zsh
echo "This script uses zsh as the interpreter!"
```

Use it or not?

If a script does not contain the shebang, the commands are executed using your shell, so there are chances that the code might run properly, but still, that isn't the correct way of doing it! Different shells have slightly varying syntax.

More than just shell scripts!

You dont have to use shell as the interpreter for your scripts. For example, you can run a python script too by supplying the path in shebang.

```
#!/usr/bin/python
print "This is a python script!"
```

To run it:

```
$ chmod 755 name.py
$ ./name.py
```

Find the code here

Variables

Variables are basically storage location that have a name and can store some data which can be changed in future.

Syntax For Variables

```
VARIABLE_NAME="Value"
```

Important

- Variables are case sensitive
- By convention, variables are uppercase
- To use a variable, just write the variable name followed by the `$` sign

Examples Of Variables

- Example 1

```
#!/bin/bash
MY_NAME="Madhav Bahl"
echo "Hello, I am $MY_NAME"
```

Download the code

- Example 2

```
#!/bin/bash
MY_NAME="Madhav Bahl"
echo "Hello, I am ${MY_NAME}"
```

Download the code

- Example 3: Assign command output to a variable

```
#!/bin/bash
CONTENTS=$(ls)
echo "The contents of this directory are: "
echo "$CONTENTS"
```

An alternative:

```
#!/bin/bash
CONTENTS=`ls`
echo "The contents of this directory are: "
echo "$CONTENTS"
```

Download the code

- Example 4

```
#!/bin/bash
SERVER_NAME=$(hostname)
echo "This script is being run on ${SERVER_NAME}"
```

Download the code

Variable Names

Alphanumeric characters, starting with an alphabet or underscore and can contain digits or underscores in between.

Valid Variable Names

- THIS3VARIABLE="ABC"
- THIS_IS_VARIABLE="ABC"
- thisIsVariable="ABC"

Invalid Variable Names

- 4Number="NUM"
- This-Is-Var="VAR"

No special character apart from underscore is allowed!

Tests

Scripts are basically needed to remove the need of again and again typing the commands which you use frequently or basically automating tasks. But, what if the script you wrote needs to execute differently under different circumstances? You can make the decisions using tests.

Syntax for tests

```
[ condition-to-test-for ]
```

Example

```
[ -e /etc/passwd ]
```

This test checks whether /etc/passwd exists, if it does, it returns true (or, it exits with a status of 0). If the file doesnt exists, it returns false (status 1).

File Test Operations

```
-d FILE_NAM  # True if FILE_NAM is a directory
-e FILE_NAM  # True if FILE_NAM exists
-f FILE_NAM  # True if FILE_NAM exists and is a regular file
-r FILE_NAM  # True if FILE_NAM is readable
-s FILE_NAM  # True if FILE_NAM exists and is not empty
-w FILE_NAM  # True if FILE_NAM has write permission
-x FILE_NAM  # True if FILE_NAM is executable
```

String Test Operations

```
-z STRING  # True if STRING is empty
-n STRING  # True if STRING is not empty
STRING1 = STRIN2 # True if strings are equal
STRING1 != STRIN2 # True if strings are not equal
```

Arithmetic Operators

```
var1 -eq var2  # True if var1 is equal to var2
var1 -ne var2  # True if var1 not equal to var2
var1 -lt var2  # True if var1 is less than var2
var1 -le var2  # True if var1 is less than or equal to var2
var1 -gt var2  # True if var1 is greater than var2
var1 -ge var2  # True if var1 is greater than or equal to var2
```

Making Decisions

Just like any script, shell scripts can make decisions based on conditions.

The IF Statement

Syntax:

```
if [ condition-is-true ]
then
  command 1
  command 2
    ...
    ...
  command N
fi
```

Example:

```bash
#!/bin/bash
SHELL_NAME="bash"

if [ "$SHELL_NAME" = "bash" ]
then
  echo "You are using bash shell"
fi
```

IF-ELSE Tree

Syntax:

```
if [ condition-is-true ]
then
  command 1
  command 2
    ...
    ...
  command N
else
  command N+1
```

```
  command N+2
    ...
    ...
  command M
fi
```

Example:

```
#!/bin/bash
SHELL_NAME="bash"

if [ "$SHELL_NAME" = "bash" ]
then
  echo "You are using bash shell"
else
  echo "You are not using the bash shell"
fi
```

IF-ELIF Ladder

Syntax:

```
if [ condition-is-true ]
then
  command 1
elif [ condition-is-true ]
then
  command 2
elif [ condition-is-true ]
then
  command 3
else
  command 4
fi
```

Example:

```
#!/bin/bash
SHELL_NAME="bash"

if [ "$SHELL_NAME" = "bash" ]
then
  echo "You are using bash shell"
elif [ "$SHELL_NAME" = "csh" ]
then
  echo "You are using csh shell"
else
  echo "You are not using the bash or csh shell"
```

```
fi
```

Case Statements

The case statements are an alternative for if statements which are a little easier to read than complex if elif ladder. However, there are some limitations to case statements.

When to use If you find yourself using an if statement to compare the same variable against some different/discrete values, you can use a case statements instead of if-elif ladder.

```
if [ "$VAR"="one" ]
then
  ...
elif [ "$VAR"="two" ]
then
  ...
elif [ "$VAR"="three" ]
then
  ...
  ...
  ...
else
  ...
fi
```

Syntax

```
case "$VAR" in
  pattern_1)
    # commands when $VAR matches pattern 1
    ;;
  pattern_2)
    # commands when $VAR matches pattern 2
    ;;
esac
```

Note We can use wildcard to create an else like statement in case.

```
case "$VAR" in
  pattern_1)
    # commands when $VAR matches pattern 1
    ;;
  pattern_2)
    # commands when $VAR matches pattern 2
    ;;
  *)
```

```
    # This will run if $VAR doesnt match any of the given patterns
    ;;
esac
```

Example

```
#!/bin/bash
read -p "Enter the answer in Y/N: " ANSWER
case "$ANSWER" in
  [yY] | [yY][eE][sS])
    echo "The Answer is Yes :)"
    ;;
  [nN] | [nN][oO])
    echo "The Answer is No :("
    ;;
  *)
    echo "Invalid Answer :/"
    ;;
esac
```

Iterative Statements

Loops can execute a block of code a number of times and are basically used for performing iterations.

The For Loop

Just like any other programming language, shell scripts also have for loops.

Syntax

```
for VARIABLE_NAME in ITEM_1 ITEM_N
do
  command 1
  command 2
    ...
    ...
  command N
done
```

```
for (( VAR=1;VAR<N;VAR++ ))
do
  command 1
  command 2
    ...
    ...
```

```
  command N
done
```

Example

```
#!/bin/bash
for COLOR in red green blue
do
  echo "The Color is: ${COLOR}"
done
```

Alternatively,

```
#!/bin/bash
COLORS="red green blue"
for COLOR in $COLORS
do
  echo "The Color is: ${COLOR}"
done
```

Example 2

In this simple example we will see how to rename each file with .txt format

```
#!/bin/bash
FILES=$(ls *.txt)
NEW="new"
for FILE in $FILES
do
  echo "Renaming $FILE to new-$FILE"
  mv $FILE $NEW-$FILE
done
```

While Loop

While loop repeats a series of commands for as long as the given condition holds true.

Syntax

```
while [ CONNDITION_IS_TRUE ]
do
  # Commands will change he entry condition
  command 1
  command 2
    ...
    ...
```

```
  command N
done
```

Infinite loop

A loop which never exits, or basically, the entry condition never becomes false. If this happens by mistake, press `ctrl+c` or `cmd+c` to exit from the running script.

```
while [ CONNDITION_IS_TRUE ]
do
  # Commands do not change the conditiono
  command 1
  command 2
    ...
    ...
  command N
done
```

However, there are some cases where you might want an infinite loop. In that case you can easily create one.

```
while true
do
  command 1
  command 2
    ...
    ...
  command N
done
```

Example: print first 10 natural numbers

```
#!/bini/bash
INDEX=1
while [ $INDEX -lt 11 ]
do
  echo "Current Number: ${INDEX}"
  ((INDEX++))
done
```

Example: Read user name until correct

```
#!/bin/bash
while [ "$CORRECT" != y ]
do
  read -p "Enter your username: " USERNAME
```

```
  read -p "Is $USERNAME correct? " CORRECT
done
```

Example: reading a file line by line

```
#!/bin/bash
LINE=1
while read CURRENT_LINE
do
  echo "${LINE}: $CURRENT_LINE"
  ((LINE++))
done < /etc/passwd
# This script loops through the file /etc/passwd line by line
```

Condition can be any test or command. If the test/command retrns a 0 exit status, it means that the condition is true and commands will be executed. If the command returns a non-zero exit status, it the loop will stop its iterations. If the condition is false initially, then the commands inside the loop will never get executed.

The continue statement

`continue` statement is used to take the flow of control to the next iteration. Any statement after continue statement is hit will not be executed and the flow of control will shit to the next iteration.

The break statement

`break` statement can be used in while loops (or other loops like for loop) to end the loops. Once the break statement is hit, the flow of control will move outside the loop.

Positional Parameters

Some arguments or parameters can be passed when we call the script.

For Example:

```
$ ./script.sh param1 param2 param3 param4
```

All the parameters will be stored in:

```
  $0 -- "script.sh"
  $1 -- "param1"
  $2 -- "param2"
  $3 -- "param3"
  $4 -- "param4"
```

Example

```
#!/bin/bash
echo "Running script: $0"
echo "Parameter 1: $1"
echo "Parameter 2: $2"
echo "Parameter 3: $3"
echo "Parameter 4: $4"
```

To access all the parameters, use `$@` sign

Example

```
#!/bin/bash
echo "Running script: $0"
for PARAM in $@
do
  echo "Parameter: $PARAM"
done
```

User Input-STDIN

`read` command accepts STDIN (Standard Input)

Syntax

```
read -p "PROMPT MESSAGE" VARIABLE
```

Example

```
#!/bin/bash
read -p "Please Enter You Name: " NAME
echo "Your Name Is: $NAME"
```

Exit Status

Every command returns an exit status, also called the return code which ranges from 0 to 255. Exit status are used for error checking.

- 0 means success
- Any code other than 0 means an error condition.

To find out what an exit status for a command means, one can look for the documentations or manual using `man` or `info` command.

`$?` contains the return code of previously executed command.

Example:

```
#!/bin/bash
ls /randomDirectory   # Any Directory which does not exist
echo "$?"    # This command will return 2
```

Another Example:

```
#!/bin/bash
HOST="google.com"
ping -c 1 $HOST     # -c is used for count, it will send the request, number of times mentio
RETURN_CODE=$?
if [ "$RETURN_CODE" -eq "0" ]
then
  echo "$HOST reachable"
else
  echo "$HOST unreachable"
fi
```

Logic Operations

Shell scripts supports **logical AND** and **logical OR**.

AND

The AND Operator results true if all the conditions are satisfied.

`&& = AND`

Example

```
#!/bin/bash
MY_VAR=10
if [ "$MY_VAR" -ge 5 ] && [ "$MY_VAR" -le 15 ]
then
  echo "Given variable is within the range"
fi
```

Note While using AND, the second command will run only if the first command returns a 0 exit status.

Exampe:

```
mkdir tempDir && cd tempDir && mkdir subTempDir
```

In this example, tempDir is created with `mkdir` command. If it succeeds, then `cd tempDir` is executed.

OR

The OR Operator results true if any one the conditions are satisfied.

`|| = OR`

Example

```
#!/bin/bash
MY_VAR=10
if [ "$MY_VAR" -ge 15 ] || [ "$MY_VAR" -le 11 ]
then
  echo "Given variable is either greater than 15 or less than 11"
fi
```

Note Just like in AND operation, here also, the first condition will be execued first. If it returns the status code 0, then next condition will not be checked.

The Semicolon

Separate commands on the same line with a semicolon in between to make sure that they all get executed. The command following the semicolon will always get executed no matter the previous command failed or succeeded.

Example:

```
mkdir dir1 ; cd dir1
```

The above code is equivalent to

```
mkdir dir1
cd dir1
```

The Exit Command

Exit command is used to expicitly define the return code. If we do not define the exit status of the shell script explicitly, then by default the exit status of the last command executed is taken as the exit status of the script. We can use exit command anywherer in the script, and as soon as exit command is encountered, the shell script will stop executing.

```
  exit 0
  exit 1
  exit 2
    ...
    ...
  exit 255
```

Example:

```
#!/bin/bash
DIR_NAME="/home/madhav/Desktop/Play"
if [ -e $DIR_NAME ]
then
  echo "Given Directory Already Exists"
  echo "Moving into ${DIR_NAME}"
  cd $DIR_NAME
```

```
  exit 0
fi
echo "Given Directory Does Not Exists"
mkdir $DIR_NAME
cd $DIR_NAME
exit 1
```

Therefore, using exit code, we can can define custom meanings to exit statuses.

Functions

A set of instructions which can be reused/called any time from the main program whenever the need for those instructions arrives. Consider a situation that you have a particular module that does a particular job, now let's suppose that job has to be done 20 (say) times in the main program (for example calculating maximum number in an array of numbers). Now, if you write the code for that 20 times, your code would become very large, however, if we write a function for that, and call that function whenever it is required, the code would remain short, easy to read and much more modularised.

Moreover, it is a great coding practise to keep your code DRY. DRY stands for Don't Repeat Yourself, i.e., write the code in such a way that you dont have to be copy and pasting same piece of code around everywhere!

Therefore, functions are great because they make the code DRY, we can write once and use that many times, using functions reduces overall length/size of script. Moreover, functions make program easier to maintain, because they make the code divided into modules making particular place to edit and troubleshoot available in case of bugs/errors.

Whenever you find yourself repeating a set of instructions, make a function for that. (A function must be defined before use).

Note: It is a good practise to define all your functions at the top before starting the main program or main instructions.

Syntax

```
function function_name() {
    command 1
    command 2
    command 3
      ...
      ...
    command N
}
```

Or,

```
function_name() {
    command 1
    command 2
    command 3
      ...
      ...
    command N
}
```

To call a function To call a function, simply write it's name on a line in the script.

```
#!/bin/bash
function myFunc () {
    echo "Shell Scripting Is Fun!"
}
myFunc
```

Note: While calling functions, do not use paranthesis like we use in other programming languages.

Calling a function from another function

To call a function from another function, simply write the function name of the function you want to call.

Example:

```
#!/bin/bash
function greetings() {
  USER=$(whoami)
  echo "Welcome ${USER}"
  currentTime
}
function currentTime() {
  echo "Current Time Is: $(date +%r)"
}
greetings
```

Positional Parameters In Functions

Just like a shell script, functions can also accept parameters.

The first parameter is stored in $1 The second parameter is stored in $2 and so on. $@ contains all the parameters.

Note: $0 is still the name of script itself, not the name of function.

To provide parameters, just write them after the function name with a white space in between.

Example

```
#!/bin/bash
function greetings() {
  for PERSON_NAME in $@
  do
    echo "Hello ${PERSON_NAME}"
  done
}
greetings World! Random Guy Stranger
```

Variable Scope

Global Variables

All variables have, by default, global scope. Having a global scope means that the value of that variable can be accessed from anywhere in the script. Variables must be defined before it is used.

Note: If a variable is defined within a function, it can not be used until that function is called at least once.

Local Variables

Local variables can be accessed only from within the function. Local variables are created using `local` keyword and only functions can have local variables. It is a good practise to keep variables inside a function local.

```
function myFunc () {
  local MY_VAR="Hello World"
  # MY_VAR is available here
  echo "$MY_VAR"
}
# MY_VAR is not available here
```

Return codes for function

A function acts just like a shell script, moreover, a function may be referred to as a shell script within a shell script. Therefore, functions also have an exit status (more precisely, a return code)

Implicitly The exit status of last command executed acts as the return code for the function.

Explicitly Return codes can me custom. `return <RETURN_CODE>`

Note Valid range for return codes = 0 to 255 (0 = Success)

We can use `$?` to get the return code of a function.

```
myFunc
echo $?
```

Wildcards

A character or a string patterns that is used to match file and directory names is/are called wildcard(s).

The process used to expand wildcard pattern into a list of files and/or directories (or basically paths) is called Globbing.

Wild Cards can be used with most of the commands that require file/dir path as argument. (Example ls,rm,cp etc).

Some Commonly Used Wildcards

*** - Matches zero or more characters   Example:**

`*.txt` `hello.*` `great*.md`

? - matches exactly one character   Example:

`?.md` `Hello?`

[] - A character class   This wildcard is used to match any of the characters included between the square brackets (Matching exactly one character).

Example: `He[loym]`, `[AIEOU]`

[!] - matches characters not included within brackets   It matches exactly one character.

Example: To match a consonant: `[!aeiou]`

Note: We can create range using characer classes

`[1-5]` – Mathces number 1 to 5

`[a-e]` – Matches character a,b,c,d,e

Predefined named character classes

- [[:alpha:]]
- [[:alnum:]]
- [[:space:]]
- [[:upper:]]]
- [[:lower:]]
- [[:digit:]]

Matching wildcard characters

In case we have to match wildcard characters themselves like `*`, or `?`, we can go with escape character - `\`

Example: `*\?` –> will match all files that end with a question mark.

Using wildcards in shell scripts

```
#!/bin/bash
# This script will backup all your .txt files in /etc/tmp
mkdir /etc/tmp
for FILE in *.txt
do
  echo "Backing up file $FILE"
  cp $FILE /etc/tmp
done
```

Logging

Logs are used to keep a record of what all happened during the execution of a shell script. Logs can store any type of information and can be used to answer the 5 W's: "Who, What, When, Where and Why".

Logs are very useful when your shell script produces a lot of output (that might scroll off your screen).

Syslog

The linux OS uses syslog standard for message logging. The Syslog standard allows programs to generate messages that can be captured, processed and stored by the system logger.

The syslog standard uses facilities and severities to caegorize messages. Each message is labelled with a facility code and a severiy level, the combinations of whose can be used to determine how the messages will be handled.

Facilities They are used to indicate what type of program or what part of the system the message originated from.

Example: kern (for kernel), user, mail, daemon, auth, local0 to local7 (for custom logs) etc.

Severities As the name suggest, they measure the severity of the message. Most severe message are emergency messages and least severe messages are the debug messages.

Example: emerg, alert, crit, err, warning, notice, info, debug

Logs are handled by system logger according to their facilities and severites and are written in a file. There are several logging rules, but they arer configurable and can be changed.

Example:

Some systems have messages stored in `/var/log/messages` or `s/var/log/syslog`

The Logger Command

The logger command generates syslog messages. `logger "Message"` By default, the logger command creates messages with user facility and notice severity.

To specify the facility and severity, use -p option followed by the facility then a `.` then the severity, and then the message.

Example:

```
logger -p local2.alert "Message"
```

Debugging

A bug is an error in a computer program/software that causes it to produce an unexpected or an incorrect result. Most of the bugs are cased by errors in the code and it's design. To fix an error, try to reach to the root of that unexpected behaviour.

The process of finding bugs in the script/program/software and fixing them is called debugging.

The bash shell provides some options that can help you in debugging your script. You can use these options by updating first line of the script.

The most popular of these options is the -x option. -x option prints commands and arguments as they execute. It is called print debugging, tracing or an x-trace.

```
#!/bin/bash -x
```

If you want to do this on the command line/terminal,

```
set -x
```

set +x to stop debugging.

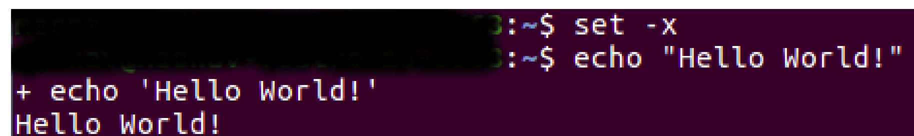

Figure 1: Preview Image

set -x will start the x-trace and set +x will stop the x-trace.

Example:

```
#!/bin/bash-x
VALUE="Hello World!"
echo "$VALUE"
```

We can also turn the debugging on for a portion of a script.

```
#!/bin/bash
echo "Turning x-trace on!"
set -x
VAL="Hello World!"
echo "$VAL"
set +x
VAL="x-trace turned off!"
echo "Turning x-trace off!"
echo "$VAL"
```

Another useful options -e = Exit on error

This will cause your script to exit immediately if a command exits with a non-zero exit status. It can be combined with other options (like the -x option) like:

```
#!/bin/bash-xe
#!/bin/bash-ex
#!/bin/bash-x-e
#!/bin/bash-e-x
```

-v = prints shell commands/input lines as they are read. It can also be combined with other options just like above.

Some Sample Programs

Click here to see the sample programs

Downloadable PDF

Claim Your Free PDF Here