Shell Scripts

Matt Warner

Contents

1 Overview

Shell scripts can do what can be done on command line

Shell scripts simplify recurring tasks. If you cannot find an existing utility to accomplish a task, you can build one using a shell script

Note:-

Much of UNIX administration and house keeping is done via shell scripts

2 Shell Script features

- Variables for storing data
- Decision-making control (e.g. if and case statements)
- Looping abilities (e.g. for and while loops)
- Functions for modularity
- Any UNIX command

  file manipulation: cat, cp, mv, ls, wc, tr, ...

  utilities: grep, sed, awk, ...
- Comments: lines starting with #

3 The basics

First line is always shebang

```
#! /bin/bash
```

To run shell scripts

```
bash script
```

Or, make executable

```
chmod +x script
./script
```

3.1 Simple Script

```
#! /bin/bash
date > usage-status
ls -l >> usage-status
du -s * >> usage-status
```

4 Bash Shell Programming Features

- Variables

  string, number, array
- input/output

  echo, printf

  command line args, read from user

- Decision

  conditional execution, if-then-else, case

- Repetition

  while, until, for

- Functions

5 User-defined shell variables

Syntax:

```
varname=value
```

Example:

```
rate=moderate
echo "Rate today: $rate"
```

Note:-

use double quotes if value of variable contains white spaces

Example:

name="Thomas William Flowers"

6 Output via echo command

- Simplest form of writing to standard output

Syntax:

```
echo [-ne] arguments
```

-n suppresses trailing newline

-e enables escape sequences:

\t horizontal tab
\b backspace
\a alert
\n newline

6.1 Examples: shell scripts with output

```
#! /bin/bash
echo "You are running these processes:"
ps
```

```
#! /bin/bash
echo -ne "Dear $USER:\nWhat's up this month:"
cal
```

7 Command line arguments

- Use arguments to modify script behavior
- command line arguments become positional parameters to shell script
- positional paramters are numbered variables

  $1, $2, $3 . . .

7.1 Meanings

$1 first parameter

$2 second parameter

${10} 10th parameter (prevents “$1” misunderstanding)

$0 name of the script

$* all positional parameters

$# the number of arguments

7.2 Example: Command Line Arguments

```
#! /bin/bash
# Usage: greetings name1 name2

echo $0 to you $1 $2
echo Today is `date`
echo Good Bye $1
```

Make sure to protect complete argument

```
#! /bin/bash
# counts lines in directory listing

ls -l "$1" | wc -l
```

If we had a bash script as such:

```
#! /bin/bash
ls -l $1 | wc -l
```

And had a file called “file example”
We would not be able to use this file as a parameter, since our argument is not protected.

8 Arithmetic expressions

Syntax:

```
$((expression))
```

This can be used for simple arithmetic

```
count=1
count=$((count+20))
echo $count
```

9 Array variables

Syntax:

```
varname=(list of words)
```

Accessed via index:

```
${varname[index]}
${varname[0]}     first word in array
${varname[*]}     all words in array
```

9.1 Using array variables

Examples

```
ml=(mary ann bruce linda dara)
echo $ml
**prints mary**

echo ${ml[*]}
**prints mary ann bruce linda dara**

echo ${ml[2]}
**prints bruce**

ml[2]=john
echo ${ml[*]}
**prints mary ann john linda dara**
```

10 Output: printf command

Syntax:

```
printf format[arguments]
```

Writes formatted arguments to standard output under the control of “format”

Format string may contain:

- plain characters: printed to output
- escape characters: e.g. \t, \n, \a . . .
- format specifiers: prints next successive arguement

10.1 printf format specifiers

```
%d  number
```

also

```
%10d    10 chars wide
%-10d   left justified
```

```
%s  string
```

also

```
%20s    20 chars wide
%-20s   left justifed
```

```
printf "random number\n"

printf "random number %d\n" 12

printf "random number %d\n" $RANDOM

printf "random number %10d\n" $RANDOM

printf "rando number %-10d %s\n" $RANDOM $USER
```

11 User input: read command

Syntax:

```
read [-p "prompt"] varname [more vars]
```

words entered by user are assigned to

varname and ”more vars”

Last variable gets rest of input file

11.1 Example: Accepting User input

```
#! /bin/bash
read -p "enter your name: "  first last

echo "First name: $first""
echo "Last name: $last"
```

Note:-

-p for prompt

12 exit Command

This terminates the current shell, the running script.

Syntax

```
exit [status]
```

Note: The default exit status is 0

Note:-

Generall convention: use negative exit status is something went wrong.

0 indicates exit success
1 indicates something minor went wrong

Predefined variable “?” holds exit status of last command

“0” indicates success, all else if failure.

Examples:

```
ls > /tmp/out
echo $?

grep -q "root" /var/log/auth.log # grep in quiet mode (exits with zero status if any match is found)
echo $?
```

13 Conditional Execution

Operators || and && allow conditonal execution

Syntax:

```
cmd1 && cmd2 # cmd2 executed if cmd1 succeeds

cmd1 || cmd2 # mcd2 executed if cmd1 fails
```

13.1 Conditional Execution: Examples

```
grep $USER /etc/passwd && echo "$USER found"

grep student /etc/group || echo "no student group"
```

14 test command

Syntax:

```
test expression
[ expression ]
```

Evaluates ‘expression’ and returns true or false

```
if test $name = "Joe"
then
  echo "Hello Joe"
fi

if [ $name = "Joe" ]
then
  echo "Hello Joe"
fi
```

15 if statements

```
if [ condition ]; then
  statements
elif [ condition ]; then
  statements
else
  statements
fi
```

15.1 test Relation Operators

Meaning	Numeric	String
Greater than	-gt	
Greater than or equal	-ge	
Less than	-lt	
Less than or equal	-le	
Equal	-eq	=
Not equal	-ne	!=
String length is zero		-z str
String length is non-zero		-n str
file1 is newer than file2		file1 -nt file2
file1 is older than file2		file1 -ot file2

Table 1: test relation operators

15.2 Compound logical expressions

`! expression`

True if `expression` is false.

`expression -a expression`

True if both `expressions` are true.

`expression -o expression`

True if one of the `expressions` is true.

15.3 Example: compound logical expressions

```
if [ ! $Years -lt 20 ]; then
  echo "You can retire now."
fi

if [ "$Status" = "H" ] && [ "$Shift" = 3 ]; then
  echo "shift $Shift gets \$$Bonus bonus"
fi

if [ "$Calls" -gt 150 ] || [ "$Closed" -gt 50 ]; then
  echo "You are entitled to a bonus"
fi
```

15.4 File Testing operators

Command	Meaning
`-d file`	true if ‘file’ is a directory
`-f file`	true if ‘file’ is a regular file
`-r file`	true if ‘file’ is readable
`-w file`	true if ‘file’ is writable
`-x file`	true if ‘file’ is executable
`-s file`	true if length of ‘file’ is nonzero

16 Debugging using “set”

The “set” command is a shell built-in command. It has options to allow tracing of execution.

-v print shell input lines as they are read

-x option displays expanded commands and its arguments

Options can be turned on or off

To turn on the option: **set -xv**

To turn off the options: **set +xv**

Options can also be set via she-bang line

```
#! /bin/bash -xv
```

Example: Using the following script

```
#!/bin/bash
read -p "enter your name: " first last
set -v
echo "First name: $first"
set +v
echo "Last name: $last"
```

This will add the following line to the output

```
echo "First name: $first"
```

Additionally, using -x will instead print the whats inside the variable.

17 The case Statement

To make decision that is based on multiple choices, we use case statements. This is can be thought of in the same way as a switch statement in C++.

Syntax:

```
case word in
  pattern1) command-list1
  ;;
  pattern2) command-list2
  ;;
  patternN) command-listN
  ;;
esac
```

Additionally, the case pattern can contain meta characters, such as:

*

?

[. . .]

[:class:]

Note:-

Multiple patterns can be listed via | (pipeline)

```
#!/bin/bash
echo "Enter Y to see all files including hidden files"
echo "Enter N to see all non-hidden files"
echo "Enter Q to quit"

read -p "Enter your choice: " reply

case "$reply" in
  Y|YES) echo "Displaying all (really...) files"
         ls -a ;;

  N|NO) echo "Displaying all non-hidden files..."
        ls ;;

  Q)    exit 0 ;;

  *) echo "Invalid choice!"; exit 1 ;;
esac
```

18 The while loop

This executes “command-list” as long as “test-command” evaluates successfully

Syntax:

```
while test-command
do
  command-list
done
```

Example:

```
#!/bin/bash
#script shows users's active processes

cont="y"
while [ "$cont" = "y" ]; do
  ps
  read -p "again (y/n)? " cont
done
echo "done"
```

Example:

```
#!/bin/bash
# copies files from home into webserver directory
# a new directory is created every hour
PICSDIR=/home/student/pics
WEBDIR=/var/www/webcam
while true; do
  DATE=`date +%Y%m%d`
  HOUR=`date +%H`
  mkdir $WEBDIR/$DATE
  while [ "$HOUR" != "00" ]; do
    mkdir $WEBDIR/$DATE/$HOUR
    mv $PICSDIR/*.jpg $WEBDIR/$DATE/$HOUR
    sleep 3688
    HOUR=`date +%H`
  done
done
```

19 The until Loop

executes “command-list” as long as
“test-command” does **not** evaluate successfully

Syntax:

```
untill test-command
do
  command-list
done
```

Example:

```
#!/bin/bash
# script shows user's active processes

stop="n"
untill [ "$stop" = "y" ]; do
  ps
  read -p "done (y/n)? " stop
done
echo "done"
```

20 The for Loop

executes ”commands” as many times as the number of words in the “word-list”

Syntax:

```
for variable in word-list
do
  commands
done
```

Example:

```
#!/bin/bash

for index in 7 6 5 4 3 2
do
  echo $index
done
```

Example:

```
#!/bin/bash
# compute average weekly temperature
TempTotal=0
for day in 1 2 3 4 5 6 7
do
  read -p "Enter temp for $day: " Temp
  TempTotal=$((TempTotal+Temp))
done
AvgTemp=$((TempTotal/7))
echo "Average temperature: " $AvgTemp
```

Note:-

instead of explicitly writting out 1 2 3 4 5 6 7

We can just write: for day in ‘seq 7‘

Or we could write: for day in Mon Tue Wed Thu Fri Sat Sun

Better yet: If we had the days of the week in a file, we could write: for day in ‘cat day-file‘

21 Looping over arguments

Simplest form will iterate over all command line arguments

Example:

```
#!/bin/bash
for parm
do
  echo $parm
done
```

22 break and continue

using the keywords **break** or **continue** will interrupt for, while or until loop

break statement:

Terminates execution of the loop

transfers control to the statement AFTER the done statement

continue statement:

skips the rest of the current iteration

continues execution of the loop

23 Shell Functions

must be defined before they can be referenced.

Note:-

Usually placed at the beginning of the script

Syntax:

```
function-name () {
  statements
}
```

Example:

```
#!/bin/bash

funky () {
  # This is a simple function
  echo "This is a funky function"
}

# declaration must precede call:

funky # function call
```

23.1 Function parameters

- Need not be declared
- Arguments provided via function call are accessible inside function as $1, $2, $3, . . .

Example:

```
#!/bin/bash
checkfile() {
  for file
  do
    if [ -f "$file" ]; then
      echo "$file is a file"
    else
      if [ -d "$file" ]; then
        echo "$file is a directory"
        fi
    fi
  done
}
checkfile . funtest
```

23.2 Local Variables in Functions

Variables defined within functions are global, i.e their values are known throughout the entire script

Keyword **local** inside a function defines variables that are local to that function, i.e not visible outside

Example:

```
#!/bin/bash
global="pretty good variable"

foo () {
  local inside="not so good variable"
  echo $global
  echo $inside
  global="better variable"
}

echo $global
foo
echo $global
echo $inside
```

23.3 return from function

Syntax:

```
return [status]
```

Ends exeuction of function

optional numeric argument sets return status
    default is “return 0”

Example:

```
#!/bin/bash
testfile() {
  if [ $# -gt 0 ]; then
    if [ ! -r $1 ]; then
      return 1
    fi
  fi
}
if testfile funtest; then
  echo "funtest is readable"
fi
```