Contents

[Overview 2](#__RefHeading__626_570918749)

[Basename function 3](#__RefHeading__628_570918749)

[System Variables 3](#__RefHeading__630_570918749)

[Command line arguments 4](#__RefHeading__632_570918749)

[User-defined Variables 6](#__RefHeading__634_570918749)

[Command line options 6](#__RefHeading__636_570918749)

[Selection statements 6](#__RefHeading__638_570918749)

[Loops (iteration) 11](#__RefHeading__640_570918749)

[Further WHILE loop experimentation. 12](#__RefHeading__642_570918749)

[Further FOR loop experimentation 14](#__RefHeading__644_570918749)

[Script Example: Backing up files 15](#__RefHeading__646_570918749)

[Using Files 17](#__RefHeading__648_570918749)

[Using Functions 18](#__RefHeading__650_570918749)

[Using Find commands 19](#__RefHeading__652_570918749)

# Overview

This worksheet introduces you to basic shell scripting on Linux / Unix (tested on both Solaris and Linux). I have not yet tested this worksheet on the RPi, so please note anything that did not work and let me know – that would be very helpful.

If you copy scripts from the web or a Word document, ensure that they do not contain ^M characters. If they do, use the utility function: dos2unix (which you might have to download), e.g.

#dos2unix \*.sh

**This will remove ^M from all files that end with .sh**

When referring to the example code supplied in this worksheet, you will see that comments in the **bash** (bourne again shell) scripting language start with #.

Shell scripting is a very useful skill, especially if you are managing Unix and Linux servers. The scripts for this worksheet are all for the **BASH** shell. Check syntax for other shell scripting languages, e.g. ksh (Korn Shell), as they differ slightly.

The **find** commands at the end of this worksheet are particularly helpful. I’ll leave you to see how…

# System Variables

Run the following commands and see what they do:

1. # ls $HOME
2. # echo $USER
3. # ls $PWD

What do the commands do?

# Setting Permissions:

If you need to add execute permissions, see the following example:

$ chmod 755 <script>. sh

|  |  |  |  |
| --- | --- | --- | --- |
| Permissions are set in blocks of 3 for owner, group and world (all other users). Use simple binary arithmetic to determine how to change the permissions. r=read, w=write, x=execute. | 421 | 421 | 421 |
| rwx | r-x | r-x |
| 4+2+1=7 | 4+1=5 | 4+1=5 |

# Command line arguments

It is often a requirement that you pass data into a script from the command line. The following section demonstrates how to do this.

Create the following script and test it with 3 command line arguments.

#!/bin/bash

# just prints out command line arguments

echo "Script Name: " $0

echo "Argument 1: " $1

echo "Argument 2: " $2

echo "Argument 3: " $3

Now try the following which shifts all arguments to the left by 1 or whatever the supplied number is, e.g. if you enter shift 2, then arguments are shifted left by 2. $0 will still be present. It can be used to skip over options, e.g. –o

Test the program by supplying command line arguments when you run it.

#!/bin/bash

shift 2

# just prints out command line arguments

echo Script: $0

Explain what you did to test the program?

Now try adding this to your script. Test with a few command line arguments.

for X in $\*

do

echo ""

echo $X

done

# User-defined Variables

The following script shows you how to create your own variables for use in a script:

#!/bin/bash

files="$(ls )"

web\_files=`ls /var/www/`

echo $files

echo "-------------------------------------------------------------"

echo $web\_files

X=`expr 3 \\* 2 + 4`

echo $X

# Command line options

Often you need to provide options as well as command line options to a program/script. For example, ls –l, ps –ef, etc. cp –R source\_file dest\_file.

Use the getopts command to get options. You will look at them in more detail after you have experimented with loops and selection statements.

# Selection statements

You’ve already used on in the Systems Variables section. You will experiment more in this section. Try the following simple if statement:

#!/bin/bash

X=""

if [ -n "$X" ]; then

echo "the variable X is not empty"

else

echo "the variable X is empty"

fi

What happens if you omit the double quotes around $X? Try this:

if [ -n $X ]; then # -n tests to see if the argument is non empty

echo "the variable X is not empty"

fi

Now try an elseif statement:

#!/bin/bash

X=2

# need space after 1 and before ]

if [ $X == 1 ]

then

echo "its 1"

elif [[ $X == 3 || $X == 2 ]]

then

echo "its not 1"

fi

Do you need to have then on a new line?

What happens if you leave out the space after the 1 and before the ] or after the [ in the if statement?

Remember to experiment to see what you can and cannot do in terms of the syntax.

Here is an example of an if statement, using –lt (less than):

#!/bin/bash

X=3

Y=4

empty\_string=""

if [ $X -lt $Y ] # is $X less than $Y ?

then

echo "\$X=${X}, which is greater than \$Y=${Y}"

fi

Experiment with the following file:

if [ -e "${HOME}/.profile" ]; then

echo "you have a .profile file"

if [ -L "${HOME}/.profile" ]; then

echo "it's a symbolic link"

elif [ -f "${HOME}/.profile" ]; then

echo "its a regular file"

fi

else

echo "you have no .profile file"

fi

Did you remember to place #!/bin/bash at the beginning of your script?

Note: the semi colon after the ‘if’ expression and before the then?

Use the read command to get data from the user when the script is running:

if [[ -z "$WELCOME" ]]; then

echo -n "Please enter your terminal type?"

read WELCOME

fi

echo $WELCOME

So, what do you think the significance of the [[ and ]] is in a script…

Try the following two examples:

1. Run script with no command line argument

if [ $1 -gt 3 ]

then

echo "$1 is more than 3"

fi

1. Again, run the following script with no command line argument

if [[ $1 -gt 3 ]]

then

echo "$1 is more than 3"

fi

How do the two scripts (a and b) compare?

Finally, you can also use a case statement in shell scripting. Try this example and experiment:

#!/bin/bash

if [[ -z "$DRINK" ]]; then

echo -n "Please enter your favourite drink? "

read DRINK

fi

echo "Your favourite drink is $DRINK"

case $DRINK in

"beer") echo "alcy";;

"coffee") echo "insomniac" ;;

"water") echo "boring";;

\*) echo "dont care about that" ;;

esac

# Loops (iteration)

Another very useful shell scripting tool is the loop. You will start with a basic WHILE loop:

#!/bin/bash -x

COUNTER=0

while [ $COUNTER -lt 10 ]; do

echo The counter is $COUNTER

let COUNTER=COUNTER+1

done

Note: the semi colon after the ] and then the word do.

Remove the –x on the /#!/bin/bash line and what happens?

Here’s a FOR loop example. This program lists command line arguments, so run it with some:

#!/bin/bash

# just prints out command line arguments

echo $0

for X in $\*

do

echo ""

echo $X

done

Now try this:

#!/bin/bash

colour1="red"

colour2="light blue"

colour3="dark green"

for X in "$colour1" "$colour2" "$colour3"

do

echo $X

done

What might you use a script like this for?

## Further WHILE loop experimentation.

#!/bin/bash

X=0

while [ $X -le 20 ]

do

echo $X

X=$((X+1))

done

echo ""

echo "again .... "

The following is same as above but using expr instead of the (( ))

X=0

while [ $X -le 20 ]

do

echo "X = $X"

#X needs to be incrememented permanently not just for printing

X=`expr $X + 1`

done

echo ""

echo "again .... "

The following script is same as above but using let instead of the (( ))

X=0

while [ $X -le 20 ]

do

echo "X = $X"

#X needs to be incremented permanently not just for printing

# needs no spaces or get error...

let X=X+1

done

# this one is not good as missing ` `. You need to ensure the syntax is correct for your scripts to run.

X=0

while [ $X -le 20 ]

do

echo $X

X= expr $X+1

done

## Further FOR loop experimentation

The following example accepts a search value as $1 to be entered at command line as first parameter, it then searches \*.txt files for the word:

#!/bin/bash

if [[ -z $1 ]]; then

echo "You forgot to pass in a command line argument"

else

{

for X in \*.txt

do

grep -i $1 "$X"

done

}

fi

Another FOR loop demo.

The script below searches for the word ‘for’ in each .txt file:

for X in \*.txt

do

grep -i "for" "$X"

done

# Script Example: Backing up files

First try this simple backup script so that you can understand what the tar utility is doing:

#!/bin/bash

OF=/volB/tr-backup-$(date +%Y%m%d).tar

tar -cf $OF /var/www/<mywebsite>/

NOTE: you need to replace <mywebsite> with the directory name of your website.

For a more advanced script, try the one below. It can be adapted for your use to backup files in a directory using the tar utility.

NOTE: IF is the input file and OF is the output file name.

#!/bin/bash

IF="/var/www/<mywebsite>"

OF="/home/adminlinux/web-backup-$(date +%Y%m%d).tgz"

### have a default IF and OF

# then if no arguments entered, use them

echo "target file is: "$OF

echo "area to backup: "$IF

echo "backing up " `date` "...."

tar -cf - $IF | gzip > $OF

error=$?

if [[ $error ]]; then

echo "SORRY, " $LOGNAME" I could not do it"

else

echo "its done, see: "

ls $OF

fi

Experiment with this script and try using it with and without command line arguments. Change the messages and add extra functionality if you wish.

# Using Files

Reading and writing to files is an essential thing for shell scripts to be able to do. Here is an example for you to experiment with:

NOTE: With this example, if you are logged in as root, you will be told file is writeable even if it is not

#!/bin/ksh

for filename in \*;

do

if [[ -d $filename ]]; then

if [[ -w $filename ]]; then

echo "I can write to this file"

ls -ld $filename

chmod -w $filename

echo "now I cannot write to this file"

fi

ls -ld $filename

fi

done

# Using Functions

Some shell scripts can become very lengthy and therefore, it is useful to be able to create modular scripts, as with most procedural programming languages. Here is an example that you can experiment with:

#!/bin/bash

drink\_menu()

{

case $DRINK in

# need command for each case, e.g. echo statement

"beer") echo "alcy";;

"coffee") echo "insomniac" ;;

"water") echo "boring";;

\*) echo "do not care about that" ;;

esac

return

}

if [[ -z "$DRINK" ]]; then

echo -n "Please enter your favourite drink? "

read DRINK

fi

#echo "Your favourite drink is $DRINK"

echo ""

drink\_menu

Note that the way you call a function in shell scripting (BASH) is to simply issue the function’s name.

Create another function and call it:

# Using Find commands

This is one of the most useful commands in Unix so it is worth your while experimenting as much as you can with the different options.

Before you become overwhelmed with the power of Unix/Linux and all it can do (there’s so much more) remember that most people don’t remember it all in their heads – they refer to manuals and notes they’ve kept. If you use commands often enough you’ll remember them, otherwise, it’s OK to keep a document of commands (your own little FAQ).

Try this set of find commands – again, experiment with them.

#!/bin/bash

banner "Find Demo"

echo " "

echo "The following command will return a list of all files in the current directory"

echo "-----------------------------------------------------------"

find . -print

echo "-----------------------------------------------------------"

echo "find . -print"

echo " "

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* "

echo " "

echo "The following command will return all directories to the current directory:"

echo "-----------------------------------------------------------"

find . -type d -print

echo "-----------------------------------------------------------"

echo "find . type -d -print"

echo " "

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* "

echo " "

echo "The following command will search the current directory, /tmp and /usr/tmp for a file with 'core' in its name. Then it will remove it"

#echo "-----------------------------------------------------------"

find . /tmp /usr/tmp -name core -exec rm {} \;

#echo "-----------------------------------------------------------"

echo "find . /tmp /usr/tmp -name core -exec rm {} \;"

echo " "

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* "

echo " "

echo "The following command will search the current directory for ".o" files, then it will ask the user if the file(s) should be removed"

#echo "-----------------------------------------------------------"

find . -name "\*.o" -ok rm {} \;

#echo "-----------------------------------------------------------"

echo "find . -name "\*.o" -ok rm {} \;"

echo " "

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* "

echo "The following command will search the entire machine for any file that is > 1k in size, then create a list in the file /tmp/large, errors will be dumped in /dev/null and the prompt will be returned immediately"

#echo "-----------------------------------------------------------"

find / -type f -size +1k -print >/tmp/large 2>/dev/null &

#echo "-----------------------------------------------------------"

echo "find / -type f -size +1k -print >/tmp/large 2>/dev/null &"

echo " "

echo "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* "

And now for a very useful command (xargs) that will save a lot of time:

#find . -type f -print | xargs grep -i <search string>

The command **xargs** reads items from the standard input (e.g. keyboard) or pipes. The data is delimited by blanks or newlines. It then executes the command one or more times with any initial-arguments followed by items read from standard input/pipe. In the above example, all files will be listed from the current directory and the output will be used as input (via the pipe | ) into the xargs command. For each item in the list (from find command), grep will be used to search for the search string (-i means, case insensitive).

Another example:

# find . -name "\*.bak" -type f -print | xargs /bin/rm –f

The above command finds all files with .bak as the extension and deletes them.

Try both of the above commands by typing them directly into the shell’s terminal window. Ensure there are some .bak files for the last example.

To create test files you can use the following:

#touch test.bak

#touch test2.bak

Now experiment with a shell script that uses find functions:

#!/bin/bash

echo "finds all files that related to the argument passed in on command line"

find . -type f -print | xargs grep -i $1

echo "this is quicker than the following version, because it builds a list of commands and runs all as a single process"

echo "finds all files that related to the argument passed in on command line"

find . -type f -exec grep -i $1 {} \;

This worksheet so far has presented many examples for you to experiment with. The following task should help you to reinforce learning and apply this knowledge to a real example.

Write a script that manages your website in the following way:

* Removes any ^M characters from all files that are copied from your external disk
* Creates a backup of your website files/directories in a tar file and appends the date to the file name.
* Moves the archive file into another directory (you can decide where)

You can implement this in any way you wish and add extra functionality.

Create further scripts to help less experienced Linux users manage your website, e.g. to find specific words in your website files. The idea is to make your website easy to manage in general.