Shell Scripting

A script is a comand line program that contains a series of ocmands.

- The commands in the script are executed by an interpreter

```
#!/bin/bash
echo "Scripting is fun!"
```

- First, the sharp + exclamation mark (bang!) are refered as Shebang, inexact contraction of "sharp bang"
- The shebang is followed by the interpreter
- The script is passed as an argument to the interpreter.
- if no shebang is specified on the first line, the script will be executed using the current shell

```
chmod 755 script1.sh
./script1.sh
>> Scripting is fun!
```

```
# Excecutes in the brackground
./sleepy.sh &
ps -fp #pid
```

You don;t have to use a shell as the interpreter for your scripts.

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

```
chmod 755 hi.py
./hi.py
This is a Python script.
```

Variables

```
VARIABLE_NAME="value"
```

- Do not use spaces before or after the equals sign.

- variables are case sensitive
- By convention, variable are in uppercase
- to use the variable precede the name with a dollar sign

```
MY_VARIABLE="MY_VARIABLE"
echo "This is my variable content $MY_VARIABLE"
```

- To precede or follow the variable with additional data you have to use curly braces

```
"${MY_VARIABLE}"
```

- If you don't encapsulte the variable name in curly braces the shell will treat the additional text as part of the variable name
- Since a vriable with that name does no texist, nothing is put in its place.

Assign Commands to a Variable

To assign the command output to a variable, enclose the comand in parentheses an precede it with a dollar sign

```
#!/bin/bash
SERVER_NAME=$(hostname)
echo "You are running this script on ${SERVER_NAME}."
```

You can also enclose the command in back ticks. This is an older syntax replaced by the $().

Variable Names

- Can contain letters, digits and underscores
- Cannot start with a digit

Tests

To make decisions when the script needs to run, we will need to test for those conditions and have the script act accordingly.

To create a test, place a conditional expression between brackets. The syntax is `[ condition-to-test-for ]`.

This test checks to see if /etc/passwd esists. If it dows, it returns true. The comand exits with a status of 0. If file does not exist, it returns false and the command exits with a status of 1.

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

In bash shell, you can run the command **help test** to see te various types of tests yoy can perform

```
File operators:
-d FILE     True if file is a directory.
-e FILE     TRUE if file exists.
-f FILE     True if file exists and is a regular file.
-r      True if file is readable by you
-s      True if file exists and is not empty
-w FILE     True if the file is writable by you
-x FILE     True if the file is executable by you

String Operators:
-z STRING       True if string is empty
- STRING        True if string is not empty
STRING      True if string is not empty
STRING1 = STRING2       True if strings are equal
STRING1 != STRING2      True if the strings are not equal

Arithmetic operators
arg1 -eq arg2
arg1 -ne arg2
arg1 -lt arg2  True if arg1 is less than arg2
arg1 -le arg2  less or equal
arg1 -gt arg2       grater than
arg1 -ge    greater or equal
```

If Statement

- The if word is followd by a test.
- The following line contains the word then

if [condition-true] then command1 command2 ... fi

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

if["$MY_SHELL"="bash"]
then
    echo "You seem to like the bash shell."
fi
```

It is a best practice t oenclose variables in quotes to prevent some unexpected effects whe nperforming conditinoal tests.

if/else

```
if [ condition-true]
then
```

```
    command1
    ...
else
    command2
    ...
fi
```

Else if

```
if[ condition-true ]
then
    ...
elif[ condition-true ]
then
    ...
else
    ...
fi
```

For

- start with the word 'for'
- Followd by a variable name
- followed by the word 'in'

```
for VARIABLE_NAME in ITEM_1 ITEM_2 ITEM_N
do
    command1
    command2
    ...
done
```

```
#!/bin/bash
for COLOR in red green blue
do
    echo "COLOR:$COLOR"
done
```

```
COLOR:red
COLOR:green
COLOR:blue
```

```
#!/bin/bash
COLORS="red green blue"

for COLOR in $COLORS
do
    echo "COLOR:$COLOR"
done
```

This shell script renames all of the files thatend in jpg by pretending today's date to the original file name

```
#!/bin/bash
PICTURES=$(ls *jpg)
DATE=$(date+%F)

for PICTURE in $PICTURES
do
    echo "Renaming ${PICTURE} to ${DATE}-${PICTURE}"
    mv ${PICTURE} ${DATE}-${PICTURE}
done
```

Positional Prameters

Positional paremeters are variables that contain the contatents of the command line. These variables are $0 trough $9. The script itself is stored in $0, the first parameter in $1, second in $2 and so on.

Comments

```
# Anything that followd the poung sign
# The only exception to this is the shebang on the first line
```

You can access all the positional parameters starting at $1 to the last one on the command line by using the special variabl $@.

```
#!/bin/bash

echo "Executing script: $0"

for USER in $@
do
    echo "Archiving user: $USER"

    # Lock the account
    passwd -l $USER

    # Create an archive of the home directory.
```

```
    tar cd /archives/${USER}.tar.gz /home/${USER}
done
```

User Input

- STD input use the `read` command.
- `read -p "PROMPT" VARIABLE_NAME`

```
#!/bin/bash

read -p "Enter a user name:" USER

echo "Archiving user: $USER"

# Lock the account
passwd -l $USER

# Create an archive of the home directory
tar cd /archives/${USER}.tar.gz /home/${USER}
```

Exist Statuses and Return Codes

Exit Status

Return code, exit code or exit status is an INTEGER RANGING from 0 to 255

- 0 -> Successfully
- Non-Zero -> Error

Error Checking

The return codes can be used in a script for error checking

```
To consult exit statuses documentation the comamnds "man" and "info" can be
useful.
```

- The special variable **$?** contains the return code of the previously executed command.

```
ls /not/here
echo "Error code: $?"
```

```
output: ls: /not/here: no such file or directory
Error code: 2
```

```
HOST="google.com"
ping -c 1 $HOST

if["$?" -eq "0"]
then
    echo "$HOST reachable."
else
    echo "$HOST unreachable."
fi
```

Chaining Commands

Operators

- & (Ampersand): Sends a process/script/command to the background
- ; (Semi-colon): The command following this operator will execute even if the command preceding is not succesfully executed.

- && (LOGICAL AND): A command following a double ampersand will only run **if the previous command exists with a 0 exit status**
- || (LOGICAL OR): The command succeeding this operator is only executed** if the command preceding it has failed**
- ! (NOT): Negates an expression within a command.
- | (Pipe): The output of the first command acts as input to the second command
- < >> > (Redirection): Redirects the output of a command or a group of commands to a file or stream
- &&-|| (AND-OR): It is a combination of AND OR operator. Similar to the if-else statement
- \ (Concatenation): Used to concatenate large commands over several lines in the shell
- () (Precedence): Allows command to execute in precedence order
- {} (Combination): The execution of the command succeeding this operator depends on the execution of the first command

& Operator

Used to run a command **in the background** -> another commands can be executed

- Increases the effective utilization of system resources and speeds up the script execution.
- This is called child process creation or forking

```
ping -cl google.com & # Change the command before &
ping -c1 google.com & ping -c1 geeksforgeeks.org &
```

&& AND Opertaor

The command succeeding this operator will only execute if the command preceding it gets succesfsfully executed. It is helpful when we want to execute a command iif the first comand has executed successfully

Piping || Opertaor

```
ls -l | wc -l
```

Seds the ouput of the first command to the input of the second command.

NOT ! Operator

```
touch a.txt b.txt c.txt d.txt e.txt
rm -r !(a.txt)
```

Negate an expression in command.

Redirection Operator

Used to redirect the ouput of a command or a group of commands to a stream or file. This operator can be used to redirect either standard input or standard output or both. Almost all comands accept input with redirection operators

```
cat >> filename
sort < filename
```

If-Else condition

```
[ ! -d ABC ] && mkdir ABC || cd ABC
```

This command will first check if the directroy ABC exists or not. If it does not exist then a new directory is created, else ABC becomes the current directory.

The exit Command

Control the exit status of your shell scripts using the exit command.

If a exit command not specify a return code with the exit command, then the exit status of the previously executed command is used as exit status

If you not include a exit command in a shell script, the exit commmand of the previously executed command is used as exit status

exit command can be used anywhere in a shell script. When a exit command is reached, the script will stop running

```
#!/bin/bash

HOST="GOOGLE.COM"
ping -c 1 $HOST
if[ "$?" -ne "0" ]
then
    echo "$HOST unreacheable."
    exit 1
fi
exit 0
```

Functions

```
#!/bin/bash

function function-name()
{
    #function code
}

function-name #calling the function
```

Positional Parameters

Functiosn can accept parameters and access ussing positional $1 $2 $3 and so on.

The $0 is still the name of the script itself

To send data to a function, supply data after the functoin name.

```
#!/bin/bash
function hello(){
    echo "Hello $1"
}
hello Mark
```

Variable Scope

- By default alla variables are global.
- The variable has to be defined before it can be used.

Local Variables

A local variable can only be accessed within the function in which it is declared. Used the `local` keyword before the variable name. The local keyword can only be used inside a function

Return Codes

Functions works like shell scripts within a shell script

The exit status in a function is called return code. If no return statement is used, then the exit status of the function is the exit status of the last command executed in the funcion.

Return statement only accepts a numbers, only integers between 0 and 255 can be used as exit status.

Backup File

The backup_file function will create a backup of a file and place it into the /var/tmp directory.

Is it recommended to use this function when a script modifies several files to make sure we have a copy to view or restore.

If it is a file and it does exist, a variable called BACKUP_FILE is created. It starts off with /var/tmp followed by the basename of the passed file, the current date and the PID of the shell script

- The **basename** command removes any leading directory componentes and returns just the file name. The basename of /etc/hosts is just hosts.
- The **date** command is using a nice format of the year, followed by the month and finally the day, all separated by dashes.
- special variable **$$** represents the PID of currently running shell script. So if you run the script multiple times on the same day PID will be different each time on the same day

```
function backup_file()
{
    if[ -f "$1" ]
    then
        local BACKUP_FILE="/var/tmp/$(basename ${1}).$(date+%F).$$"
        echo "Backing up $1 to ${BACKUP_FILE}"
        cp $1 $BACKUP_FILE
    fi
}

backup_file /etc/hosts

if[ $? -eq 0 ]
then
    echo "Backup succeeded!"
fi
```

Wildcards

A wildcard is a character or a string that is used to match file and directory names.

If a command accepts a file or directory as an argument, you can use a wildcard in the argument to specify a file or set of files.

- **asterisk**: '*'
- **Question Mark**: '?'

Asterisk matches zero or more characters. It matches anything. For example, you could use '* .txt' to find all the files that end in .txt. Or to list all the files that start with the letter 'a' using 'a *'.

Question mark matches exactly one character. ?.txt matches all the file that have only one character preceding a txt.

Character Classes

A character classes is used to create specific search patterns

- Starts with a left bracket
- List one or more characters
- End with right bracket

```
c[au]t # matches for cat or cut file

[!aeiou]* # ecludes first letter vowel files

[a-g] or [1-8] # Ranges
```

Named Character Classes

Predefined Character Classes. Represent the most commonly used ranges.

```
[:alpha:] # Matches alphabetic, lower and uppercase

[:alnum:] # Matches alphanumeric characters.

[:digit:] # numbers, 0 to 9

[:lower:]

[:upper:]

[:space:] # Matches whitespace, spaces, tabs and newline characters
```

Matching Wildcard Patterns

To match one of the wildcard characters then yoi would escape that character with a backslash.

It is a good practice not naming files with question marks and asterisks.

Wildcard in Shell Scrippts

```
#!/bin/bash
cd /var/www
cp *.html /var/www-just-html
```

```
#!/bin/bash
cd /var/www
for FILE in *.html
do
    echo "Copying $FILE"
    cp $FILE /var/www-just-html
done
```

Case Statements

- Starts with the word case followed with an expression or variable and end the line with the word in
    -
- list a pattern or value you want to test against the variable. End the pattern with a parenthesis
    - `pattern_N)`
- End the commands with a double semicolon.
- End with `esac` word

Example 1: Basic

```
case "$1" in
    start)
        /usr/sbin/sshd
    ;;
    stop)
        kill $(cat /var/run/sshd.pid)
    ;;
esac
```

Example 2: Wildcards

```
case "$1" in
    start|START)
        /usr/sbin/sshd
        ;;
    stop|STOP)
        kill $(cat /var/run/sshd.pid)
        ;;
    *)
        echo "Usage: $0 start|stop"
        exit 1
        ;;
esac
```

```
read -p "Enter y or n:" ANSWER
case "$ANSWER" in
    [yY]|[yY][eE][sS])
        echo "You answered yes."
    ;;
    [nN]|[nN][oO])
```

```
        echo "You answered no."
esac
```

Logging

Keep a record of what ocurred during the execution of a shell script with a logging mechanism. Logs can store any type of infromation you want, but they typically answer who, what, when, where, and why something ocurred.

Linux uses syslog standard for message logging. This allows programms and apps to generate messages that can be captured, processed and stored by the system logger. It elminates the need for each and every app having to implement a logging mechanism. That means we can take advantage of this logging system in our shell scripts.

Syslog standard uses facilities and severities to categorize messaages. Each msg is labeled with a facility code and a severity level. Combination of facilities and severities can be used to determine how a message is handled.

Facilities are used to indicate what type of program or what part of the system the message originated from. Messages that are labeled wit h the **kern** facility originate from the Linux kernel. Messages that are labeled with the **mail** facility come from applications involved in handling mail.

There are several facilities. If your script is involved in handling mail you could use the user facilityu. Also, the facilities ranging from local0 to local7 are to be used to create custom logs. These facilities would also be appropiate for custom written shell scripts.

```
Number Keyword Description
0   kern    kernel messages
1   user    user level messages
2   mail    mail system
3   daemon  styystem daemons
4   auth    security/authorization messages
5   syslog  messages generated by syslogd
6   lpr     line printer subsystem
7   news    networks news subsystem
8   uucp    uucp subsystem
9   clock   daemon
10  authpriv
11  ftp
12  ntp
```

The severities are emergency, alert, critical, error, warning, notice, info, and debug.

```
Code Severity Keyword Description
0   Emergency   emerg(panic)    System is Unusable
1   Alert       alert           Action must be taken inmediately
2   Critical    crit            Critical conditions
3   Error       error(err)      Error conditions
4   Warning     warning(warn)   Warning conditions
5   Notice      notice          Normal but significant condition
```

```
6   Info        info            Informational messages
7   Debug       debug           Debug level messages
```

Each Linux distro uses a slightly sifferent set of defaults, and logging rules are configurable and can be changed.

Many messages are stored in `/var/log/messages` or `/var/log/syslog`.

Logger

Logger command generate syslog messages. In its simplest form you simply supply a message to the logger utility. By derfault, the logger utility creates messages using the user facility and the notice severity

The message generated without options includes date, user and message.

```
logger "Message"
logger -p local0.info "Message" # Uses Local0 Facility
logger -t myscript -p local0.info "Message" # Tag Message
logger -i -t myscript "Message" # Process ID (PID)
logger -s -p local0.info "Message" # -s: Displayed on Screen
```

Create a function in shell script to handle logging

```
logit()
{
    local LOG_LEVEL=$1
    shift
    MSG=$@
    TIMESTAMP=$(date +"%Y-%m-%d %T")
    if [ $LOG_LEVEL = 'ERROR' ] || $VERBOSE
    then
        echo "${TIMESTAMP} ${HOST} ${PROGRAM_NAME} [${PID}]: ${$LOG_LEVEL}
${MSG}"
}
```

- Logit expects that log level followed by a message passed into it.
- `shift` command is run to shift the positional parameters to the left.
- If the log level is error or the VERBOSE global variable is set to true, a message is echoed to the sccreen, which includes info such timestamp, log level and the message.

Debugging

- The process of finding errors in your script or fixing unexpected behaviors is called debugging.

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

-x Option

The -x option prints commands and their arguments as they are executed. This menas that instead of variables being displayed, the values of those values are displayed. The same thing goes for expansions. Wildcards aren't displayed, but what the expand to is displayed. You-ll sometimes hear this type of debugging called 'print debugging', 'tracing' or 'x-trace'.

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

On the command line

```
set -x # Starts debugging behavior
set +x # Stops  debuggin behavior

# These options can also used for a portion of the script
set -x
    # Debugging Behavior Section
set +x
```

-e Option

It causes your script to exit immediately if a command exits with a non-zero status.

```
#!/bin/bash -xe
set -xe
set +xe
```

-v Option

Prints the shell commands just like they are read from the script. Prints everything before any subtitutions and expansions are applied.

The combination -xv to see what line looks like before and after substitutions and expansions occur

```
#!/bin/vash -v
TEST_VAR="test"
```

```
echo "$TEST_VAR"
```

Output

```
#!/bin/bash -vx
TEST_VAR="test"
+ TEST_VAR=test
echo "$TEST_VAR"
+ echo test
>> test
```

set Command

From command line you can run `help set`.

Many times using -x, -e and/or -v is sufficient. But if you want a bit more control over debugging you can create your own codee to do it. One method is to create a variable called DEBUG, set to true if currently debugging or false if not.

Booleans

Bash built-in booleans are true and false. To use a boolean do not quote them.

Syntax Highlighting

Checklist

1. Starts with a shebang?
2. Include a comment describing the purpose?
3. Are the global variables declared at the top of the script?
4. Are all functions grouped together following global variables?
5. Functions uses local variables? ```local GREETING="Hello!"
6. The main body of shell script follow the functions?
7. Script exits with an explicit exit status?
8. Are exit statuses explicitly used at varios exit points?

Template

```
#!/bin/bash
#
# Code Description

GLOBAL_VAR1="one"
GLOBAL_VAR2="two"

function function_one(){
    local LOCAL_VAR1="one"
}

# Main Body

# Exit with an explicit exit status
exit 0
```