Shell Scripting

Shebang

- shebang is located at the top of your script and is followed by the path to an interpreter

  Example: **#!/bin/bash for bash shell**

- It indicates which interpreter to use for the commands listed in the script. The interpreter being what executes commands in your script

---

Execute the script

- chmod +x filename.sh
- ./filename.sh
- sh filename.sh
- bash filename.sh

Variables

- Variables are case sensitive and by convention are uppercase. Be sure to not use any space around the "=" sign. Example: **VARIABLE_NAME="value"**
- to use variable in your script use the $ sign.

  Example: name="linux"

  echo " current os is $name "

File operators (tests)

- -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 FILE: True if file is readable by you
- -s FILE: True if file exists and is not empty
- -w FILE: True if file is writable by you
- -x FILE: True if file is executable by you

Arithmetic operators (tests)

- arg1 -eq arg2: True if arg1 is equal to arg2
- arg1 -ne arg2: True if arg1 is not equal to arg2
- arg1 -lt arg2: True if arg1 is less than arg2
- arg1 -le arg2: True if arg1 is less than or equal to arg2
- arg1 -gt arg2: True if arg1 is greater than arg2
- arg1 -ge arg2: True if arg1 is greater or equal to arg2

If, elif and else statement syntax

```
if [ condition-if-true ]
then
   command 1
```

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

For loop syntax

```
for VARIABLE_NAME in ITEM_1 ITEM_N
do
   command 1
   command 2
   command N
Done
```

Example: For loop

```
#!/bin/bash
COLORS=”red green blue”
for COLOR in $COLORS
do
   echo “COLOR: $COLOR”
done
```

Positional parameters

Give parameters when executing the script

$ script.sh parameter1 parameter2 parameter3… can be found with:

$0: “script.sh”

$1: “parameter1”

$2: “parameter2”

$3: “parameter3”

Example:

#!/bin/bash

echo “Executing script: $0”

echo “Archiving user: $1”

Accepting User Input

read -p user-input

Example:

#!/bin/bash

read -p “Enter a user name: “ USER

echo “archiving user: $USER”

Exit statuses and return code

- every command returns an exit status which range from 0 to 255
- 0 = success ; other means error condition which is used for error checking
- use **man** or **info** to find meaning of exit status
- $? contains the return code of the previously executed command

---

Logical

- && = AND. second command will be run only if the first command succeeded
- || = OR. second command won’t be run if the first command succeeded
- **semicolon**: separate commands to ensure they **all** get executed

---

Exit command

- explicitly define the return code. Default value to the last command executed
- When the **exit** command is reached, **your script will stop running**
- examples: exit 0, exit 2, exit 255

Example:

```
#!/bin/bash

HOST=”google.com”

ping -c 1 $HOST

if [ “$?” -ne “0” ]

then

   echo “$HOST unreachable”

   exit 1

fi

exit 0
```

Functions

```
function hello() {

   echo “Hello!”

}
```

- note: you call the function **without parenthesis**, i.e hello
- if you need parameters you do: hello param_1

Case statements

```
case “$VAR” in
   pattern_1)
      commands_go_here
      ;;
   pattern_N)
      commands_go_here
      ;;
esac
```

Example:

```
case “$1” in
   start)
      /usr/sbin/sshd
      ;;
   stop)
      kill $(cat /var/run/sshd.pid
      ;;
   *)
      echo “XXXXXX”
```

```
    ;;
esac
```

While loop

```
while [ condition_is_true ]
do
   command 1
done
```

Example:

```
INDEX=1
while [ $INDEX -lt 6 ]
do
   echo “XXXXX”
   ((INDEX++))
done
```

Break and continue

You can also use the **break** and **continue** statement inside a loop to control when the loop should stop

- **break**: exit a loop before the normal ending

- **continue**: restart the loop at the next iteration before the loop completes

---

Debugging

Built in debugging help

- **-x = prints commands as they execute**; so arguments are printed as they are executed
- after substitutions and expansions
- called an x-trace, tracing, or print debugging
- **#!/bin/bash -x**
- in the script: set -x to start debugging. Set +x to stop debugging

Example:

#!/bin/bash

TEST_VAR='test'

set -x

echo $TEST_VAR

set +x

- **-e = exit on error**
- can be combined with other option: #!/bin/bas -ex

Example:

#!/bin/bash -e

FILE_NAME='/not/here'

ls $FILE_NAME

echo $FILE_NAME

- **-v = print shell input as they are read**
- can be combined with other options

Example:

#!/bin/bash -v

TEST_NAME='test'

echo $TEST_NAME

Standard Input, Output, & Error

- **STDIN** (0) — Standard input (data fed into the program)
- **STDOUT** (1) — Standard output (data printed by the program, defaults to the terminal)
- **STDERR** (2) — Standard error (for error messages, also defaults to the terminal)

Save the output (STDOUT) to a file with > or >>

"The greater than operator (>) indicates to the command line that we wish the programs output (or whatever it sends to STDOUT) to be saved in a file instead of printed to the screen"

ls > test.txt

By default, it will create a new file or if the file already exists, clear its content and save the new output. If we need to **append** the output to a file, use >>

ls > >test.txt

Feed the input of a program (STDIN) with a file by using <

"Read data from the file and feed it into the program via it's stream".

Here wc counts the number of words in myoutput

wc -l < myoutput

wc -l < barry.txt > myoutput

Redirect STDERR

You can use numbers to indicate to save STDERR into a file. STDERR is represented by number 2

ls -l video.mpg blah.foo 2> errors.txt

In the example above, if there is an error, the message will be saved into errors.txt.

ls -l video.mpg blah.foo > myoutput 2>&1

Send data from one program to another with |

This is called piping.

ls | head -3

ls | head -3 | tail -1

Sed

Sed is used to search for a particular string in a file and then apply diverse operations such as replace, delete, insert, etc… So **you can edit a file without opening it. It is often used for string replacement**

sed “s/[Cc]omputer/COMPUTER/g” file

For example, here we are doing a substitution (indicated by s/) changing all occurrences (indicated by g for global) of computer or Computer by COMPUTER in the file indicated

---

Grep

“Global Regular Expression Print or grep is a command-line tool which is basically used to search for a string of characters in a specified file. The grep filter **searches a file for a particular pattern of characters, and displays all lines that contain that pattern**”

On the contrary to sed, it is used mainly to return lines from a file

grep “literal_string” filename

Awk

“This command searches for text-based files or data and is basically **used for generating information or manipulating data**. It also allows users to implement numeric functions, string functions, logical operators, etc. It is **useful for the transformation of data files** along with creating formatted reports.” It is used for pattern scanning and processing

awk ‘/manager/ {print}’ employee.txt

For example, it prints all lines from the employee.txt file where we have the word “manager”

awk ‘{print 4}’ employee.txt

Print the word 1 and 4 of each line
