Computer Practicum 1

Introduction to Bash scripts

Vida Groznik

Source: Ryan‘s tutorials.

What is a Bash script?

- A script tells a computer what it should do or say.
- With a Bash script we are telling the Bash shell what it should do.
- A Bash script is a **plain text file** which contains a **series of commands**.
  - Anything you can run normally on the command line can be put into a script and it will do exactly the same thing.
- By convention we give the Bash script files an extension **.sh**.

How can I run a Bash script?

- Before running a script we have to set the execute permission.
  (for safety reasons this permission is generally not set by default).
- To run the script, type in:

```
./scriptName.sh
```

```
user@bash: ./script.sh
bash: ./script.sh: Permission denied
user@bash: chmod 755 script.sh
user@bash: ./script.sh
Hello world!
```

- If you just type a name on the command line, Bash tries to find it in a series of directories stored in a variable called $PATH and **doesn't consider sub directories or your current directory.**
  - We can see the current value of this variable using the command **echo.**
- If a script is not in one of the directories in your $PATH then you can run it by telling Bash where it should look to find it.
- You can include an absolute or relative path in front of the program or script name.
  - **The dot (.)** is a reference to your **current directory**. Assuming this script is in the home directory you can also run it by using an absolute path.

How does a Bash script look like?

```
1. #!/bin/bash
2. # A sample Bash script
3.
4. echo Hello World!
```

- **Line 1** - Is what's referred to as the **shebang**.
  - **#!/bin/bash** is the first line of the script.
    The hash exclamation mark (**#!**) character sequence is referred to as the Shebang. Following it is the path to the interpreter that should be used to run (or interpret) the rest of the lines in the text file.
    (For Bash scripts it will be the path to Bash.)
  - The shebang must be on the **first line** of the file (line 2 won't do, even if the first line is blank).
  - There are **no spaces** before the **#** or between the **!** and the path to the interpreter.

- **Line 2** - This is a **comment**. Anything after **#** is not executed. It is for our reference only.
- **Line 4** - Is the command echo which will print a message to the screen.

Variables

- A variable is a temporary store for a piece of information. There are two actions we may perform for variables:
  - **Setting a value** for a variable.
  - **Reading the value** for a variable.
- To read the variable we place its name (preceded by a $ sign) anywhere in the script we would like.
- Before Bash interprets every line of our script it first checks to see if any variable names are present.
- For every variable it has identified, it replaces the variable name with its value. Then it runs that line of code and begins the process again on the next line.
- Syntax:
  - When **referring to or reading a variable** we **place a $ sign** before the variable name.
  - When **setting a variable** we **leave out the $** sign.
  - Some people write variable names in uppercase so they stand out. It's your preference.

Command line arguments

```
1. #!/bin/bash
2. # A sample Bash script
3.
4. cp $1 $2
5.
6. # Let‘s verify the values
7.
8. echo Details for $2
9. ls -l $2
```

- To supply arguments to a script, we use the variables **$1** to represent the **first command line argument**, **$2** to represent the **second command line argument** and so on.
- **Line 4** - run the command **`cp`** with the first command line argument as the source and the second command line argument as the destination.
- **Line 8** - run the command **echo** to print a message.
- **Line 9** - After the copy has completed, run the command **`ls`** for the destination just to verify it worked. We have included the options **`l`** to show us extra information so we can verify it copied correctly.

Special variables

`$0` - The name of the Bash script.

`$1 - $9` - The first 9 arguments to the Bash script.

`$#` - How many arguments were passed to the Bash script.

`$@` - All the arguments supplied to the Bash script.

`$?` - The exit status of the most recently run process.

`$$` - The process ID of the current script.

`$USER` - The username of the user running the script.

`$HOSTNAME` - The hostname of the machine the script is running on.

`$SECONDS` - The number of seconds since the script was started.

`$RANDOM` - Returns a different random number each time is it referred to.

`$LINENO` - Returns the current line number in the Bash script.

If you type the command **`env`** on the command line you will see a listing of other variables which you may also refer to.

Setting our own variables

```
1. #!/bin/bash
2. # A simple variable example
3.
4. myvariable=Hello
5.
6. anothervar=Mike
7.
8. echo $myvariable $anothervar
9. echo
10.
11. sampledir=/etc
12.
13. ls $sampledir
```

We may also set our own variables. There are a few ways in which variables may be set but this is the basic form: `variable=value`

- There is **no space** on either side of the **equals (=) sign**. We also leave the $ sign from the beginning of the variable name when setting it.

- **Lines 4 and 6** - set the value of the two variables `myvariable` and `anothervar`.
- **Line 8** - run the command **echo** to check the variables.
- **Line 9** - run the command **echo** without arguments to get a blank line on the screen.
- **Line 11** - set a variable as the path to a particular directory.
- **Line 13** - run the command **`ls`** substituting the value of the variable `sampledir` as its first command line argument.

Exercise

Perform number calculation in bash script and store result to third variable, lets say a=5, b=8, c=a+b?

Perform number calculation in bash script and store result to third variable. Put the values of the numbers as an argument to a script.

Quotes

```
user@bash: myvar='Hello World'
user@bash: echo $myvar
Hello World
user@bash: newvar="More $myvar"
user@bash: echo $newvar
More Hello World
user@bash: newvar='More $myvar'
user@bash: echo $newvar
More $myvar
```

When we want variables to store **more complex values**, we need to make use of **quotes**. Under normal circumstances Bash uses a **space to determine separate items**.

- When we enclose our content in quotes we are indicating to Bash that the contents should be considered as a single item. You may use single quotes (') or double quotes (").
- **Single quotes** will treat every character **literally**.
- **Double quotes** will **allow** you to do **substitution** (that is include variables within the setting of the value).

Command substitution

```
1. user@bash: ls
2. bin Documents Desktop ...
3. Downloads public_html ...
4. user@bash: myvar=$( ls )
5. user@bash: echo $myvar
6. bin Documents Desktop Downloads
   public_html ...
```

Command substitution allows us to take the **output** of a command or program (what would normally be printed to the screen) and **save it as the value** of a variable. To do this we place it **within brackets, preceded by a $ sign**.

- Command substitution is simple if the output of the command is a single word or line. If the output goes over several lines then the newlines are simply removed and all the output is in a single line.
- **Line 1** - Run the command **`ls`**. Normally its output would be over several lines.
- **Line 4** - When we save the command to the variable **`myvar`** all the newlines are stripped out and the output is all on a single line.

User input

```
1. #!/bin/bash
2. # Ask the user for their name
3.
4. echo Hello, who am I talking to?
5.
6. read varname
7.
8. echo It\'s nice to meet you $varname
```

```
user@bash: ./introduction.sh
Hello, who am I talking to?
Vida
It's nice to meet you Vida
user@bash:
```

Use a command called **read**, to ask the user for an input. It takes the input and saves it into a variable.

```
read var1
```

- **Line 4** - Print a message asking the user for input.
- **Line 6** - Run the command **read** and save the users response into the variable **varname.**
- **Line 8** - **echo** another message to verify the read command worked.

User input (2)

```
#!/bin/bash
# Ask the user for login details

read -p 'Username: ' uservar
read -sp 'Password: ' passvar
echo
echo Thank you $uservar we now have your login details
```

```
user@bash: ./login.sh
Username: vida
Password:
Thank you vida we now have your login
details
user@bash:
```

To alter the behaviour of **read** there are a variety of command line options. Two commonly used options are:

-p to specify a prompt and

-s makes the input silent.

This can make it easy to ask for a username and password combination (see example).

User input (3)

```bash
#!/bin/bash
# Demonstrate how read works

echo What cars do you like?
read car1 car2 car3

echo Your first car was: $car1
echo Your second car was: $car2
echo Your third car was: $car3
```

```
user@bash: ./cars.sh
What cars do you like?
Jaguar Maserati Bentley Lotus
Your first car was: Jaguar
Your second car was: Maserati
Your third car was: Bentley Lotus
user@bash:
```

We can supply several variable names to **read**. It will take the input and split it on whitespaces.

- The first item will be assigned to the first variable name, the second item to the second variable name and so on.
- If there are **more items than** variable **names**, the remaining items will **all** be added to the **last variable** name.
- If there are **less** items than variable names, the remaining variable names will be set to **blank or null**.

Reading from STDIN

```
1. #!/bin/bash
2. # A basic summary of my sales report
3.
4. echo Here is a summary of the sales data:
5. echo ==============================
6. echo
7.
8. cat /dev/stdin | cut -d' ' -f 2,3 | sort
```

It's common in Linux to **pipe** a series of simple, single purpose commands together to create a larger solution tailored to our exact needs. We can use this mechanism with our scripts as well.

- Bash accomodates piping and redirection using special files.
- Each process gets it's own set of files (one for STDIN, STDOUT and STDERR) and they are linked when piping or redirection is invoked :
  - **STDIN** - `/dev/stdin` or `/proc/self/fd/0`
  - **STDOUT** - `/dev/stdout` or `/proc/self/fd/1`
  - **STDERR** - `/dev/stderr` or `/proc/self/fd/2`
- **Line 8** - **cat** the file representing STDIN, **cut** setting the delimiter to a space, fields 2 and 3 then sort the output.

Reading from STDIN (2)

```bash
#!/bin/bash
# A basic summary of my sales report

echo Here is a summary of the sales data:
echo ==============================
echo

cat /dev/stdin | cut -d' ' -f 2,3 | sort
```

```
user@bash: cat salesdata.txt
Fred apples 20 November 4
Susy oranges 5 November 7
Mark watermelons 12 November 10
Terry peaches 7 November 15
user@bash:
user@bash: cat salesdata.txt | ./summary
Here is a summary of the sales data:
====================================

apples 20
oranges 5
peaches 7
watermelons 12
user@bash:
```

Arithmetic: function **let**

```
1. #!/bin/bash
2. # Basic arithmetic using let
3.
4. let a=5+4
5. echo $a # 9
6.
7. let "a = 5 + 4"
8. echo $a # 9
9.
10. let a++
11. echo $a # 10
12.
13. let "a = 4 * 5"
14. echo $a # 20
15.
16. let "a = $1 + 30"
17. echo $a # 30 + first command line argument
```

`let` is a built-in function of Bash that allows us to do simple arithmetic.

```
let <arithmetic expression>
```

As you can see in the example, it can take a variety of formats. The **first part is a variable** which the result is saved into.

- **Line 4** - This is the basic format. If we don't put quotes around the expression, it must be written without spaces.
- **Line 7** – Quotes allow us to use space in the expression to make it more readable.
- **Line 10** – A shorthand for increment the value of the variable a by 1.
  It is the same as writing "a = a + 1".
- **Line 16** - We may also include other variables in the expression.

Arithmetic: basic expressions

Operator	Operation
+	addition
-	subtraction
/*	miltiply
/	divide
var++	increase the variable var by 1
var--	decrease the variable var by 1
%	modulus (return the remainder after division)

Arithmetic: function **expr**

```
1. #!/bin/bash
2. # Basic arithmetic using expr
3.
4. expr 5 + 4 # 9
5.
6. expr "5 + 4" # 5 + 4
7.
8. expr 5+4 # 5+4
9.
10. Expr 11 % 2 # 1
11.
12. expr 5 \* $1 # result of (5 * first arg.)
13.
14. a=$( expr 10 - 3 )
15. echo $a # 7
```

expr is similar to `let` except instead of saving the result to a variable it **prints out** the answer.

- You don't need to enclose the expression in quotes.
- Use spaces between the items of the expression.
- It is common to use **expr** within command substitution to save the output to a variable.

```
expr item1 operator item2
```

- **Line 4** - The basic format. Note that there must be spaces between the items and no quotes.
- **Line 6** - If we put quotes around the expression, the expression will be printed.
- **Line 8** - If we do not put spaces between the items of the expression, the expression will be printed.
- **Line 10** - Here we demonstrate the operator **modulus**.
- **Line 12** - Some characters have a special meaning to Bash so we must escape them (put a backslash in front of) to remove their special meaning.
- **Line 14** - This time we're using expr within command substitution in order to save the result to the variable **a**.

Arithmetic: double parentheses

```
1.  #!/bin/bash
2.  # Basic arithmetic using double parentheses
3.
4.  a=$(( 4 + 5 ))
5.  echo $a # 9
6.
7.  a=$((3+5))
8.  echo $a # 8
9.
10. b=$(( a + 3 ))
11. echo $b # 11
12.
13. b=$(( $a + 4 ))
14. echo $b # 12
15.
16. (( b++ ))
17. echo $b # 13
18.
19. (( b += 3 ))
20. echo $b # 16
21.
22. a=$(( 4 * 5 ))
23. echo $a # 20
```

To save the output of the basic arithmetic, we can use double brackets:

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

- **Line 4** - This is the basic format. As you can see we may space it out nicely for readability without the need for quotes.
- **Line 7** - It works just the same if we take spacing out.
- **Line 10** – We can include variables without preceding $ sign.
- **Line 13** - Variables can be included with the $ sign if you prefer.
- **Line 16** - Here the value of the variable b is incremented by 1 When we do this we don't need the $ sign before the brackets.
- **Line 19** - A slightly different form of the previous example. Here the value of the variable b is incremented by 3.
- **Line 22** - Unlike other methods, when we do multiplication we don't need to escape the * sign.

Length of a variable

```bash
#!/bin/bash
# Show the length of a variable.

a='Hello World'
echo ${#a} # 11

b=4953
echo ${#b} # 4
```

To find out the length of a variable (how many characters) we can use:

`${#variable}`

IF statements

```
1. #!/bin/bash
2. # Basic if statement
3.
4. if [ $1 -gt 100 ]
5. then
6.      echo Hey that\'s a large number.
7.      pwd
8. fi
9.
10. date
```

```
user@bash: ./if_example.sh 15
Mon 6 Nov 17:20:40 2017
user@bash: ./if_example.sh 150
Hey that's a large number.
/home/vida/bin
Mon 6 Nov 17:20:40 2017
user@bash:
```

A basic `if` statement says, if a particular test is true, then perform a given set of actions. If it is not true then don't perform those actions.

```
if [ <some test> ]
then
        <commands>
fi
```

Anything between `then` and `fi` (if backwards) will be executed only if the test (between squared brackets) is true.

- **Line 4** - If the first cmd. line argument is greater than 100.
- **Line 6 and 7** - Will only run if the test on line 4 returns true. You can have as many commands here as you like.
- **Line 6** - The backslash (\ ) in front of the single quote (') is needed as the single quote has a special meaning for bash and we don't want that special meaning.
- **Line 8** - `fi` signals the end of the if statement. All commands after this will be run as normal.
- **Line 10** - This command will be run regardless of the outcome of the if statement.

Test command

The square brackets (`[ ]`) in the `if` statement are actually a reference to the command `test`. This means that all of the operators that test allows may be used here as well. Some of the more common operators are listed below.

Operator	Description
`! EXPRESSION`	The EXPRESSION is false.
`-n STRING`	The length of STRING is greater than zero.
`-z STRING`	The lengh of STRING is zero (ie it is empty).
`STRING1 = STRING2`	STRING1 is equal to STRING2
`STRING1 != STRING2`	STRING1 is not equal to STRING2
`INTEGER1 -eq INTEGER2`	INTEGER1 is numerically equal to INTEGER2
`INTEGER1 -gt INTEGER2`	INTEGER1 is numerically greater than INTEGER2
`INTEGER1 -lt INTEGER2`	INTEGER1 is numerically less than INTEGER2
`-d FILE`	FILE exists and is a directory.
`-e FILE`	FILE exists.
`-r FILE`	FILE exists and the read permission is granted.
`-s FILE`	FILE exists and it's size is greater than zero (ie it is not empty).
`-w FILE`	FILE exists and the write permission is granted.
`-x FILE`	FILE exists and the execute permission is granted.

NOTE:

- **=** is slightly different to **-eq**.
  [001 = 1] will return false as = does a string comparison (ie. character for character the same) whereas -eq does a numerical comparison meaning [001 -eq 1] will return true.
- When we refer to FILE we are actually meaning a path. Remember that a path may be absolute or relative and may refer to a file or a directory.
- Because [ ] is just a reference to the command **test** we may experiment and trouble shoot with test on the command line to make sure our understanding of its behaviour is correct.

Test command (2)

```
1.  user@bash: test 001 = 1
2.  user@bash: echo $?
3.  1
4.  user@bash: test 001 -eq 1
5.  user@bash: echo $?
6.  0
7.  user@bash: touch myfile
8.  user@bash: test -s myfile
9.  user@bash: echo $?
10. 1
11. user@bash: ls /etc > myfile
12. user@bash: test -s myfile
13. user@bash: echo $?
14. 0
15. user@bash:
```

- **Line 1** - Perform a string based comparison. Test doesn't print the result so instead we check it's exit status which is what we will do on the next line.
- **Line 2** - The variable **`$?`** holds the exit status of the previously run command (in this case test). **`0` means TRUE (or success). 1 = FALSE (or failure).**
- **Line 4** - This time we are performing a numerical comparison.
- **Line 7** - Create a new blank file **`myfile`** (assuming that `myfile` doesn't already exist).
- **Line 8** - Is the size of `myfile` greater than zero?
- **Line 11** - Redirect some content into `myfile` so it's size is greater than zero.
- **Line 12** - Test the size of `myfile` again. This time it is TRUE.

Nested if statements

```
1.  #!/bin/bash
2.  # Nested if statements
3.
4.  if [ $1 -gt 100 ]
5.  then
6.     echo Hey that\'s a large number.
7.
8.     if (( $1 % 2 == 0 ))
9.     then
10.        echo And is also an even number.
11.    fi
12. fi
```

You may have as many **`if`** statements as necessary inside your script. It is also possible to have an `if` statement inside of another `if` statement.

- **Line 4** - Perform the following, only if the first command line argument is greater than 100.
- **Line 8** - This is a light variation on the if statement. If we would like to check an expression then we may use the double brackets just like we did for variables.
- **Line 10** - Only gets run if both if statements are true.

If - else statement

```
#!/bin/bash
# else example
# read from a file if it is supplied as
# a command line argument, else read
# from STDIN.

if [ $# -eq 1 ]
then
    nl $1
else
    nl /dev/stdin
fi
```

Sometimes we want to perform a certain set of actions if a statement is true, and another set of actions if it is false. We can accommodate this with the `else` mechanism.

```
if [ <some test> ]
then
        <commands>
else
        <other commands>
fi
```

Exercise

Write a bash script that will add two numbers, which are supplied as command line argument, and if this two numbers are not given show error and its usage.

Exercise

Write script to determine whether given file exist or not, file name is supplied as command line argument, also check for sufficient number of command line argument.

If – elif – else statement

Example: if you are 18 or over you may go to the party. If you aren't but you have a letter from your parents you may go but must be back before midnight. Otherwise you cannot go.

```bash
#!/bin/bash
# elif statements

if [ $1 -ge 18 ]
then
    echo You may go to the party.
elif [ $2 == 'yes' ]
then
   echo You may go to the party but be back before midnight.
else
   echo You may not go to the party.
fi
```

Sometimes we may have a series of conditions that may lead to different paths.

```
if [ <some test> ]
then
        <commands>
elif [ <some test> ]
then
        <different commands>
else
        <other commands>
fi
```

You can have as many `elif` branches as you like. The final `else` is also optional.

Exercise

Write Script to find out biggest number from given three numbers. Numbers are supplies as command line argument. Print error if sufficient arguments are not supplied.