



ATMS 305 WEEK 2: INTRODUCTION TO UNIX/LINUX

Lecture 3: bash variables and
scripting

Adapted from: <http://ryanstutorials.net/bash-scripting-tutorial/bash-script.php>

BASH SCRIPTING

Last week, we introduced the bash shell as a way to create files, start a program to edit them, move around in the file system, display directory and file contents, and change file permissions.

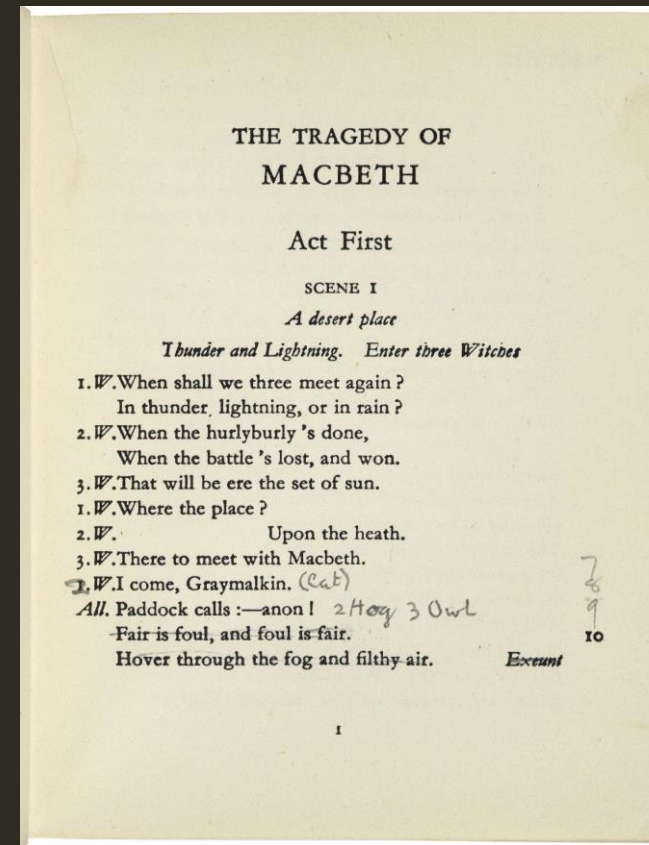
This week we will learn about scripting in **bash**.

WHAT IS SCRIPTING?

Scripting is nothing new — it is just a file that tells the computer exactly what to do by issuing a sequence of commands.

We will use our knowledge of bash and unix commands to write scripts to do automated tasks.

This is quite useful if you have a large number of tasks to do, or you want to do something automatically at a specific time, etc.



HOW DO WE RUN (EXECUTE) SCRIPTS?

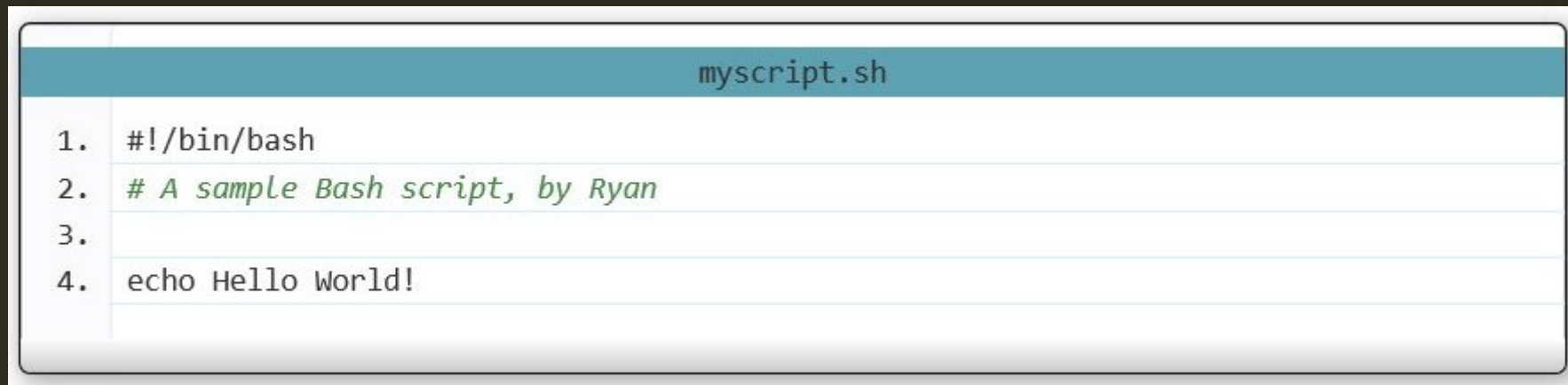
At the command line. But permissions must be execute for the user that wishes to use them!

```
Terminal
1. user@bash: ./myscript.sh
2. bash: ./myscript.sh: Permission denied
3. user@bash: ls -l myscript.sh
4. -rw-r--r-- 18 ryan users 4096 Feb 17 09:12 myscript.sh
5. user@bash: chmod 755 myscript.sh
6. user@bash: ls -l myscript.sh
7. -rwxr-xr-x 18 ryan users 4096 Feb 17 09:12 myscript.sh
8. user@bash: ./myscript.sh
9. Hello World!
10. user@bash:
```

OUR FIRST SCRIPT

We can use a text editor to create a script. We use sh as a file extension for bash scripts, so we can tell what kind of file it is.

Here is an example script:



```
myscript.sh
1. #!/bin/bash
2. # A sample Bash script, by Ryan
3.
4. echo Hello World!
```

```
myscript.sh
1. #!/bin/bash
2. # A sample Bash script, by Ryan
3.
4. echo Hello World!
```

- **Line 1** - Is what's referred to as the **shebang**. See below for what this is.
- **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. You can type this command yourself on the command line and it will behave exactly the same.
- The syntax highlighting is there only to make it easier to read and is not something you need to do in your own files (remember they are just plain text files).

WHY THE ./ ?

You've possibly noticed that when we run a normal command (such as `ls`) we just type its name but when running the script above I put a `./` in front of it. When 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`. We can see the current value of this variable using the command `echo` (you'll learn more about variables later).



```
Terminal
1. user@bash: echo $PATH
2. /home/ryan/bin:/usr/local/bin:/usr/bin:/bin
3. user@bash:
```

THE SHEBANG (#!)

#!/bin/bash

This is the first line of the script above. The hash exclamation mark (`#!`) character sequence is referred to as the Shebang. Following it is the path to the interpreter (or program) 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, but there are many other types of scripts and they each have their own interpreter.)

Formatting is important here. The shebang must be on the very first line of the file (line 2 won't do, even if the first line is blank). There must also be no spaces before the `#` or between the `!` and the path to the interpreter.

VARIABLES IN BASH

Variables are temporary stores of information in the bash shell (and are useful in scripts)

We will learn how to:

- Store information in a variable
- Retrieve information from a variable that has been stored

VARIABLE TIPS

When referring to or reading a variable we place a \$ sign before the variable name.

```
echo $PATH
```

When setting a variable we leave out the \$ sign.

```
TARGET="/home/snesbitt"
```

Some people like to always write variable names in uppercase so they stand out. It's your preference however. They can be all uppercase, all lowercase, or a mixture.

A variable may be placed anywhere in a script (or on the command line for that matter) and, when run, Bash will replace it with the value of the variable. This is made possible as the substitution is done before the command is run.

SPECIAL VARIABLES

Command line arguments: useful for passing information into a script.

```
mycopy.sh
1. #!/bin/bash
2. # A simple copy script
3.
4. cp $1 $2
5.
6. # Let's verify the copy worked
7.
8. echo Details for $2
9. ls -lh $2
```

```
mycopy.sh
1.  #!/bin/bash
2.  # A simple copy script
3.
4.  cp $1 $2
5.
6.  # Let's verify the copy worked
7.
8.  echo Details for $2
9.  ls -lh $2
```

- 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 and **h** to make the size human readable so we may verify it copied correctly.

OTHER SPECIAL VARIABLES

\$0 - The name of the Bash script.

\$1 - \$9 - The first 9 arguments to the Bash script. (As mentioned above.)

\$# - 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 it is referred to.

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

env – All the current variables!

SETTING OUR OWN VARIABLES

simplevariables.sh

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

simplevariables.sh

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

Terminal

```
1. user@bash: ./simplevariables.sh
2. Hello Fred
3.
4. a2ps.cfg aliases alsa.d ...
5. user@bash:
```

QUOTES

A terminal window with a yellow title bar labeled "Terminal". The window has a light yellow background. It shows a list of three items, each preceded by a number. The first item is a command, the second is an error message, and the third is a prompt.

```
1. user@bash: myvar=Hello World
2. -bash: World: command not found
3. user@bash:
```

Bash uses spaces to separate items.

Quotes will keep items together.


```
Terminal
1. user@bash: myvar='Hello World'
2. user@bash: echo $myvar
3. Hello World
4. user@bash: newvar="More $myvar"
5. user@bash: echo $newvar
6. More Hello World
7. user@bash: newvar='More $myvar'
8. user@bash: echo $newvar
9. More $myvar
10. user@bash:
```

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

A terminal window titled "Terminal" with a yellow background. It shows a sequence of seven numbered commands and their outputs. The commands demonstrate how to capture the output of a command into a variable using 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 ...
7. user@bash:
```

Note the spaces!!

In a script, use the echo command to test before running script!

script1.sh

```
1. #!/bin/bash
2. # demonstrate variable scope 1.
3.
4. var1=blah
5. var2=foo
6.
7. # Let's verify their current value
8.
9. echo $0 :: var1 : $var1, var2 : $var2
10.
11. export var1
12. ./script2.sh
13.
14. # Let's see what they are now
15.
16. echo $0 :: var1 : $var1, var2 : $var2
```

script2.sh

```
1. #!/bin/bash
2. # demonstrate variable scope 2
3.
4. # Let's verify their current value
5.
6. echo $0 :: var1 : $var1, var2 : $var2
7.
8. # Let's change their values
9.
10. var1=flop
11. var2=bleh
12.
```

```
Terminal
1. user@bash: ./script1.sh
2. script1.sh :: var1 : blah, var2 : foo
3. script2.sh :: var1 : blah, var2 :
4. script1.sh :: var1 : blah, var2 : foo
5. user@bash:
```

Note that the exported variable is propagated into the script!

SUMMARY

Stuff We Learnt

\$1, \$2, ...

The first, second, etc command line arguments to the script.

variable=value

To set a value for a variable. Remember, no spaces on either side of =

Quotes " '

Double will do variable substitution, single will not.

variable=\$(command)

Save the output of a command into a variable

export var1

Make the variable var1 available to child processes.

Also: Spaces are important!!