OS1 Lab 07 - Linux Shell Scripting & Environment Variables

Dr. Martin O’Connor

The purpose of this lab activity is to introduce Unix/Linux Shell Scripting and Environment Variables. This lab activity introduces new concepts unfamilar to you. Thus, there are full demonstrations provided for several example exercises. Please work through and complete all the exercises below (including the examples) using the Google Cloud Shell Linux Terminal.

As you are working through this lab session, don’t forget to copy the contents of all the scripts you create (both from the examples and the tasks) into a word document, to be shown to the Teach Assistant before leaving the lab.

## The Unix/Linux Shell

The command line interface (cli) in Unix/Linux is called a shell and consists of both an interactive command language as well as a scripting language. In UNIX/Linux there are five well known shells:

1. The Bourne Shell (sh). This was one of the early Unix shells first released in 1979 and the ancestor of most modern shells.
2. **The Bourne Again Shell (bash). This is a Unix shell written as a free GNU software replaced for the Bourne shell. It was first released in 1989. It has become the default login shell for most Linux distributions. Bash is a POSIX-compliant shell with support for a number of additional extensions. We use the bash shell in this module.**
3. The C Shell (csh) [and the improved version Turbo C Shell (tcsh)]. The C shell was originally written by Bill Joy (the original author of the VI editor) in 1978. The design goal of the C shell was a command language that was more stylistically consistent with the rest of the Unix OS system which was written in the C programming language.
4. The Korn Shell (ksh). This is an early shell developed at Bell Lab and first released in 1983. The Korn shell is backward compatible with the Bourne Shell and includes may features from the C shell.
5. The Z Shell. The Z shell is an extended Bourne Shell with many improvements incorporated from the bash, ksh, and tcsh shells. It was first released in 1990. In 2019, Apple changed the default shell in MacOS desktops and laptops from the Bourne Again shell (bash) to the Z Shell (zsh). **Oh My Zsh** [[1]](#footnote-1) is an online open source, community-driven framework that collects and makes available third-party pluggins and themes that significantly enhance the power and functionality of the zsh shell.

### Shell Prompt

As mentioned earlier, in this module we focus on the bash shell.

In the bash shell, when logged in as a user, the default prompt presents the $ (dollar) symbol. Thus, the typical Google Cloud Shell Linux terminal prompt will appear like:

username@cloudshell:~ (project-name) $

If you are using the C Shell, the default prompt is the % symbol:

username@machine:~%

On most Linux and Unix systems when you are logged in as the superuser *root*, the default prompt changes to the # (hash or pound) symbol:

root@machine:~#

The *root* superuser has almost unlimited privileges in a Unix/Linux OS. A mistyped command executed with *root* privileges can do perhaps untold damage to a computer system. That is why the prompt is changed by default to the # (hash or pound) symbol to make it clear to the user to be careful!

The Google Cloud shell is an exception to the above rule concerning the superuser *root* and the # (hash or pound) symbol. However, it makes sense given that the Google Cloud Shell is ephemeral and no changes to the underlying system persist beyond the next reboot/restart.

### Bash Shell Features

The Bash shell provides many useful features and functionality, including the following:

* Editing and Automated Command Line Completion.
* History and Command Re-entry
* Job Control
* Shell Functions and Aliases
* Arrays
* Arithmetic
* Tilde Expansion
* Brace Expansion
* Substring Capabilities
* Indirect Variable Expansion
* Expanded I/O Capabilities
* Control of Builtin Commands
* Help
* Shell Optional Behaviour
* Prompt Customization
* Security

If you would like more information about each of the above Bash shell features, please visit the [Case Western Bash Page](https://tiswww.case.edu/php/chet/bash/bash-intro.html).

## Shell Scripts

We can create batch files in Unix/Linux which allow us to write multiple commands together into a single file called a script. Those multiple commands can be executed by running the script. Shell scripting allows us to combine complex and repetitive sequences of commands into one single script which can be stored and run at any time.

Indeed, a Unix/Linux shell script is a fully-fledged programming language in itself. In a shell script, one can define local and environment variables, create user-defined functions, evaluate conditional expressions (if statements), perform loop operations (for/while loops), switch the flow of control (case statements), process command line arguments, read input from a user/file/terminal, write output to terminals/processes/files, process arrays and so on.

## Getting started with Shell Scripts

To begin your journey learning Linux scripting, consider the basic concept of a shell script to be a simple list of commands, which will be run in the order they are listed. A good shell script will often have comments before each block of commands outlining the steps to be performed. Please note that shell scripts are interpreted at runtime (like python and ruby programs). This means they are not compiled (like C or C++ programs).

## How to Create an Hello World Shell Script using the VIM text editor.

To create a shell script using the Vim editor:

1. In the Linux terminal, type the command

* $ vim myscript.sh

1. Enter the following text into the script

* #!/bin/bash  
  # This is a comment!  
  echo Hello OS1

1. Save the file and exit the Vim editor. (the commands to save a file in Vim were demonstrated in a previous lab session).

Given that a script is a file intended to be executed, the script filename should always have the extension **.sh** The purpose of this extension is to aid humans in identifying the file as a script. Technically, a Unix/Linux script does not require an extension or indeed a script can have any extension. However, it is convention and standard practice to use the **.sh** extension and this convention should always be followed, unless there is a good and compelling reason not to follow it.

## Explanation of the Hello World Shell Script

The first line in the shell script is:

#!/bin/bash

This line informs the Unix/Linux Operating System that the script is to be executed using the Bash shell (even if the current shell is not the bash shell). To be precise, the first line specifies the absolute path to the shell that must execute the script. In this case, the absolute path is /bin/bash.

This first line is very common in Unix/Linux scripts, so much so, that it has earned it own name - the **Shebang**. The Shebang is an *interpreter directive*, that is to say, it directs the current shell to use the specified interpreter (shell/scripting language/program) to parse and execute the script. The Shebang is often the first line in the script.

The Shebang is the character sequence consisting of the pound sign character **#** followed by an exclamation mark **!** followed by the absolute path to the interpreter (in this case /bin/bash).

The second line of the script is:

# This is a comment!

The hash symbol at the beginning of the second or subsequent lines indicates the line is a comment. These lines are completely ignored by the shell. A hash symbol may appear at any position on a line, and anything that appears after the hash symbol until the end of the line is treated as a comment.

The third line of the script is:

echo Hello OS1

This line contains just one command to be executed echo hello OS1 which simply echos (prints) the given string to stdout on the terminal.

## How to run/execute a Unix/Linux Script

To execute a shell script:

1. First, the file must be executable. To make the file executable, you must grant the execute permission to whomever needs to run the file. In this case, grant yourself (the owner of the file) the execute permission on the script. Type the command:

* $ chmod u+x myscript.sh

1. To run the script, type the command:

* $ ./myscript.sh
* The script runs and displays the line:
* Hello OS1
* Congratulations! you just ran your first script.
* Note: in order to run the script, you had to type the prefix **./** before the script. This prefix informs the shell that the script is located in the current working folder. By default, Unix/Linux does not look/search in the current working folder for a program or script. Instead, it searches the **Path** environment variable (to be discussed later). In Unix/Linux, you have to explicitly tell the command line terminal where a user-created script or program is located in order to run it.

## Example 2

In this example, we demonstrate further usage of a comment and also attempt to run the script without the execute permission enabled to see what happens.

1. Create a new script called myscript2.sh

* $ vim myscript2.sh

1. Enter the following text into the script

* #!/bin/bash  
  # This is a comment!  
  echo My second script # This is a comment too!

1. Save the file and exit the Vim editor.
2. Try to run the script. Type the command:

* $ ./myscript2.sh
* The following error message is displayed by the shell:
* -bash: ./myscript2.sh: Permission denied
* You need to allow execute permission on the script.

1. Grant execute permission to the owner of the script.

* $ chmod u+x myscript2.sh

1. Try to run the script once more. Type the command:

* $ ./myscript2.sh
* The script will run and the following message should be displayed:
* My second script

## Example 3

In this example, we will write a script that will create a variable and display its value.

1. Create a new script called myscript3.sh

* $ vim myscript3.sh

1. Enter the following text into the script

* #!/bin/bash  
  MESSAGE="Hello OS1 Group 1B"  
  echo $MESSAGE
* Note 1: There should be NO space either side of the equals sign.  
  Note 2: A variable may only take one value, so a string containing spaces must be quoted.

1. Save the file and exit the Vim editor.
2. Grant execute permission to the owner of the script.

* $ chmod u+x myscript3.sh

1. To run the script, type the command:

* $ ./myscript3.sh
* The following message is displayed:
* Hello OS1 Group 1B

## Example 4

In this example, we will write a script that interactively sets a variable and display its value.

1. Create a new script called myscript4.sh

* $ vim myscript4.sh

1. Enter the following text into the script

* #!/bin/bash  
  echo What is your name?  
  read NAME  
  echo "Hello $NAME - I hope you are enjoying this lab session."

1. Save the file and exit the Vim editor.
2. Grant execute permission to the owner of the script.

* $ chmod u+x myscript4.sh

1. To run the script, type the command:

* $ ./myscript4.sh
* Assuming you entered Martin as the name, the following is displayed:
* What is your name?  
  Martin  
  Hello Martin, I hope you are enjoying this lab session.

## 

## Environment (Shell) Variables

Variables are basically “boxes” that hold values. You may need to create variables for many reasons. You may need them to hold user input, arguments, or numerical values. Take for instance the following piece of code

#!/bin/bash  
X=12  
echo "The value of variable X is $X"

What you have done here, is to give X the value of 12. The line echo "The value of variable X is $X" prints the current value of X. When you define a variable, it must not have any whitespace before of after the assignment operator **=**. Here is the syntax:

variable\_name=this\_value

The value of a variable is accessed by prefixing the variable name with the dollar symbol **$**. In the example above, we accessed the value of X by using echo $X.

Please note the following points:

* The bash shell gets unhappy if you leave a space on either side of the assignment operator **=**. For example, the following gives an error message (the underscore \_ denotes a space):
* $ X\_=\_hello
* It is not necessary to always use quotes when you are assigning a value to a variable. However you do need quotes if the value includes spaces (such as a string/sentence).
* $ X=hello world # Error  
   $ X="hello world" # Ok

## Global Environment Variables

Global Environment variables are set by the system. These variables typically store information about the shell session and the working environment. Environment variables are a feature that allows a programmer to store data in memory that can be easily accessed by any program or script running from the shell. This is a handy way to store persistent data that identifies features of the user account, system, shell, or anything else you need to store.

To obtain a listing of all environment variables in your shell, run the command:

$ env  
OR  
$ printenv

As you can see, there are many environment variables that are set for the shell. Most of them are set by the system during the login process. There are often too many to view on screen at one time. Thus, run the command again, but this time pipe the output to the viewer less:

$ env | less  
OR  
$ printenv | less

Within the viewer less, pressing enter moves the content down one line at a time; pressing spacebar moves the content down one page at a time. Recall that the viewer less is effectively like running the Vim editor in read-only mode. Thus, many of the Vim editor nagivation commands work identically in the viewer less. To find help within the less view, press **h**. To quit the help information page, press **q**. To quite the viewer less press **q**.

To display the value of an individual environment variable, use the echo command. Recall, to access the value of an environment variable, you must place a dollar sign before the environment variable name.

For example, run each of the following commands and observe the output for the most commonly used environment variables in Linux:

$ echo $SHELL # Prints the absolute path of the current shell  
$ echo $USER # Prints the current logged in username  
$ echo $PWD # Prints the absolute path of the current working folder  
$ echo $PATH # Prints all directories on the Path  
$ echo $HOME # Prints the absolute path of the user home folder  
$ echo $HOSTNAME # Prints the name of the host computer  
$ echo $LANG # Prints the default system encoding  
$ echo $EDITOR # Prints the default file editor

## Local Environment Variables

Local Environment variables are only available in the current shell. The built-in command

$ set

without any options (or arguments) will display a list of all variables and functions in the current shell.

You may define a variable as follows:

$ X="hello"

Then to reference the value of this local environment variable, prefix it with the **$** symbol. For example:

$ echo $X

## Single and Double Quotations

Everything between the two quotes **“** and **“** is taken literally (the same as in Python). Therefore, if you have the following line in your script:

echo "printing X to the screen"

you will see the following:

printing X to the screen

In addition, everything between **’** and **’** is taken literally, except for another **’**. Single quotation marks will always give you exactly what’s inside the single quotation marks - any characters that might otherwise have special meaning to the shell (like the dollar sign or the backslash) are treated literally (i.e. not treated as being special).

Always use double quotation marks when you want to assign a string that contains special characters the shell should act on. The following are the most common special characters and their special meaning:

* $   variable substitution will occur.
* `   command substitution will occur (the key beside 1 on your keyboard. This will sometimes enter 2 single quotes so delete 1).
* \   The backslash is used to escape (treat literally) a single character (such as $ or \*) that might otherwise be treated as a special character by the shell. That is to say, the character following a \ is taken literally.

## Task 1

Run the following examples in the Linux command line terminal and write the output displayed as your answer.

1. Exercise 1

* $ X="hello world"  
  $ echo "printing $X to screen"
* Answer:

1. Exercise 2

* $ echo \$X
* Answer:

1. Exercise 3

* $ echo 'Single quotes "protect" double quotes'
* Answer:

1. Exercise 4

* $ echo "Well, isn't that \"special\"?"
* Answer:

1. Exercise 5

* $ echo "You have the following files in `pwd`"
* Answer:

1. Exercise 6

* $ X=100  
  $ echo "The value of \$X is $X"
* Answer:

## Task 2

What is the outcome of the following commands?

1. Exercise 1 (note: single quotes)

* $ Greeting='Good Morning $USER'  
  $ echo $Greeting
* Answer:

1. Exercise 2 (note: double quotes)

* $ Greeting="Good Morning $USER"  
  $ echo $Greeting
* Answer:

## Task 3

The output of the following commands demonstrates an error. What is the error and how do you fix it?

$ cost="Price is $5.00"  
$ echo $cost

Answer:

## Bash Script Variables

The following variables are available and accessible from inside a bash script.

* $0 - The name of the Bash script.
* $1 - $9 - The first 9 arguments to the Bash script.
* $# - The number of arguments passed to the Bash script when it was run.
* $@ - All the arguments supplied to the Bash script.
* $$ - The process ID of the current script.

## Task 4

Write a script that takes one command line argument - your first name - and displays the message (assume the name Joe was given):

Hi Joe, don't you just love this module!

Answer:

## Task 5

Write a script that takes two command line arguments, your first name and your favourite colour, and displays the message (assume the name Joe and the colour red were given):

Joe prefers the colour red.

Answer:

## Task 6

Write a script that may receive an arbitrary number of arguments (between 0 and 9 inclusive). The script must do the following:

* print the name of the bash script itself.
* print the username of the user running the script.
* print the number of arguments received.
* print all the arguments received.
* print the process ID of the running bash script.
* print the current working folder.

Answer:

## Bonus Exercise

Create a shell script called linecount.sh which outputs your login name and the number of files in your current directory, followed by the name of your current directory.

Hint:

* You should store the number of files in the current directory in a variable called *FILECOUNT*.
* To determine the number of files in the current directory, you may use redirection in combination with the commands wc, ls, and grep. OR
* To determine the number of files in the current directory, you may use redirection in combination with the commands wc and find.
* To learn more about the commands wc, ls, grep, and find, look up the system help by using the man command

Answer:

1. <https://ohmyz.sh/> [↑](#footnote-ref-1)