Beginners/BashScripting

Bash scripting is one of the easiest types of scripting to learn, and is best compared to Windows Batch scripting. Bash is very flexible, and has many advanced features that you won't see in batch scripts.

However if you are a 'non-computer-savvy' person that won't mean a thing to you. Bash is the language that you will learn to love as much of everyday Ubuntu life is done/can be done using the Terminal. You will soon learn that most things can be done through both GUI (Graphical User Interface) and CLI (Command Line Interface), however some things are more easily achieved from one or the other. For example, changing file permissions of a folder and all its sub folders is more easily achieved using cli instead gui.

NOTE: Text that is inside the box are to be entered into a terminal as follows:

```
If it's inside a box like this... enter it into a terminal unless instructed to do otherwise. Except this box. Of course. Silly.
```

You can also just copy and paste if needed.

Contents

1. Intro
2. Bash -- Everyday Ubuntu life
3. Commands
    1. Creating folders
    2. Deleting files
    3. Deleting folders and their sub-folders
    4. Running commands as root
    5. Opening GUI applications as root
    6. Backing up your files
    7. Backing up your Directories
    8. Checking system performance
    9. Check Devices
    10. Show network information
    11. Show wireless information
4. Scripting
    1. Variables
    2. If Statements
5. Storing application stdout to a variable:
    1. Example
    2. Example 2
6. FUNctions
    1. Example
        1. Debugging
7. Other Scripting Languages related to Bash
    1. tr
        1. Example
        2. Example
    2. AWK
        1. pidof clone
    3. SED
        1. Basic Substitution

Intro

In this document we will discuss useful everyday commands, as well as going a bit more in depth into scripting and semi-advanced features of Bash. Bash is not only used to run programs and applications, but it can also be used to write programs or scripts.

Bash -- Everyday Ubuntu life

During your time as an Ubuntu user you will use the terminal to perform tasks such as

- Creating folders
- Deleting files
- Deleting folders and their sub-folders
- Opening applications as root
- Backing up your files
- Backing up your folders
- Checking system performance
- Check Devices
- Checking wireless connection

Along with many other things, the list above will be the commands we will discuss.

Commands

Creating folders

Creating folders can be done simply in the file manager nautilus by right clicking and selecting 'Create Folder', but if you want to do this from a cli environment you would type the following in the terminal:

```
mkdir /home/joe/Desktop/new_folder
```

the *mkdir* (make directory) command creates the folder then the file path tells it where to create the folder.

Deleting files

Deleting files are done with the rm command as follows:

```
rm /home/joe/file_to_be_deleted
```

the *rm* (remove) command is used to remove anything through a cli environment.

Deleting folders and their sub-folders

The command you are about to read can potentially (if used incorrectly) **destroy** your system!

```
rm -rf /home/joe/useless_Parent_folder
```

This command is slightly different to the one before, it uses two options '-r' which means recursive (will delete the folder and all sub-folders) and '-f' means force (will not ask for your permission). This command is perfectly fine for deleting a dir and all its sub-dirs. The next commands should **!!**!!NEVER!!**!!** be run. Unless you want to say goodbye to your system.

```
rm -rf /*
rm -rf /
```

This will delete everything from your root folder downwards, which if you did a standard install would be **everything**.

Running commands as root

When working on the command line, you usually want to work with the default permissions. This way you insure that you won't accidentally break anything belonging to the system or other users, so long as the system and other users haven't altered their file permissions. Yet there will be times when you wish to copy a file to a system folder (like /usr/local/bin) to make it available to all users of the system. Only the system administrator (i.e. the user 'root') should have permission to alter the contents of system directories like /usr/local/bin. Therefore trying to copy a file (like a program downloaded from the Internet) into that folder is forbidden by default.

```
cp Downloads/some_downloaded_program /usr/local/bin
cp: cannot create regular file `/usr/local/bin/some_downloaded_program': Permission denied
```

Since it would be very tedious to always login as root to do administrative work (in fact you should avoid logging in as root with a graphical desktop) you can use the *sudo* program to execute a single command as root.

```
sudo cp Downloads/some_downloaded_program /usr/local/bin
```

The default Ubuntu installation is configured so that the user who was created during system installation is allowed to use this command. It will prompt for the password and execute the command as the root user.

Opening GUI applications as root

Sometimes you will want to edit a config file in your root folder, in order to save changes to this file you need root privileges so we need to open our text editor as root.

(assuming your text editor is gedit)

```
gksudo gedit /path/to/conf_file.txt
```

gksudo is the same as *sudo* but should be used to open any graphical applications as root while *sudo* is intended for executing single commands. Serious damage can be done to your system by editing these files. It is advised to create a backup of any file you edit.

Backing up your files

To create a backup of a file, we're going to use the *cp* (copy) command. The basic syntax for *cp* is as follows:

```
cp source_file dest_file
```

This will copy the 'source_file' to the 'dest_file'. Now, using the previous example, we want to backup '/path/to/conf_file.txt'. To accomplish this, we type the following:

```
sudo cp /path/to/conf_file.txt /path/to/conf_file.txt.old
```

That's fine and dandy, but what if I want to copy my file to another directory? Well that's just as easy. Let's say instead of copying /path/to/conf_file.txt to the /path/to/ directory, you want to copy it to a directory where you store all of your backup files, say /my/backup/folder/. In order to accomplish this you would type:

```
cp /path/to/conf_file.txt /my/backup/folder/    #saves conf_file.txt to /my/backup/folder/
#OR
```

```
cp /path/to/conf_file.txt /my/backup/folder/conf_file_new_name.txt
```

This is a typical safety measure that has saved many users in the past from a complete disaster.

Okay, so we know how to copy a file: a) to a different filename and b) to a different folder. But how do we copy entire directories?

Backing up your Directories

To backup one directory to another, we introduce *cp -r* (recursive) option. The basic syntax is as follow:

```
cp -r /directory/to/be/copied/ /where/to/copy/to/
```

So if we wanted to copy all of the contents of our /path/to/ folder to our /my/backup/folder, we would type the following:

```
cp -r /path/to/ /my/backup/folder/foldername    #foldername can be whatever you want the foldername to be
```

Checking system performance

If your computer starts to lag, you can see which applications are using the most CPU power with this command:

```
top
```

This is generally the same information given as the GUI application 'System Monitor'.

Check Devices

USB Devices If a USB device stops working, you may want to see if it is still connected/detected. To check if a device is connected/detected, type the following:

```
lsusb
```

PCI Devices PCI devices are checked with:

```
lspci
```

Show network Information

To see the status of your network connection, use the command:

```
ip addr
```

This command will show you your ip, what type of connection you are using, etc.

Show wireless information

Like the command ip stated above, you can use iwconfig to check the settings of your wireless connection without editing anything. In a terminal enter:

```
iwconfig
```

This also shows packets sent/received.

Scripting

NOTE: The commands given in the scripting section are to be put into the text editor and not in the terminal unless instructed otherwise.

Bash is primarily a scripting language, so it would be a crime not to talk about scripting. Let's dive straight in with a bash script. More precisely the infamous "Hello World" script. You can create a bash script by opening your favorite text editor to edit your script and then saving it (typically the .sh file extension is used for your reference, but is not necessary. In our examples, we will be using the .sh extension).

```
 #!/bin/bash

echo "Hello, World"
```

The first line of the script just defines which interpreter to use. NOTE: There is no leading whitespace before #!/bin/bash. That's it, simple as that. To run a bash script you first have to have the correct file permissions. We do this with chmod command in terminal (change mode) as follows:

```
chmod a+x /where/i/saved/it/hello_world.sh   #Gives everyone execute permissions
# OR
chmod 700 /where/i/saved/it/hello_world.sh   #Gives read,write,execute permissions to the Owner
```

This will give the file the appropriate permissions so that it can be executed. **Now open a terminal and run the script like this**:

```
/where/i/saved/it/hello_world.sh
```

Hopefully you should have seen it print Hello, World onto your screen. If so well done! That is your first Bash script.

TIP If you type:

```
pwd
```

You will see the directory that you are currently working in (*pwd* stands for 'print working directory'). If your current working directory is /where /i/saved/it/, then you can shorten the above command to:

```
prompt$ pwd
/where/i/saved/it
prompt$ ./hello_world.sh
```

Now, lets get to more interesting aspects of Bash programming, Variables!

Variables

Variables basically store information. You set variables like this using text editor:

```
var="FOO"
```

'var' can be anything you want as long as it doesn't begin with a number. "FOO" can be anything you want.

To access the information from the variable you need to put a '$' in front of it like this:

```
var="FOO"
echo $var
```

Try entering those lines into a terminal one at a time; you will see that the first one just gives you another prompt and the second one prints FOO.

But that's all a bit boring. So let's make a script to ask the user for some information and then echo that information.

```
 #!/bin/bash
clear
echo "Please enter your name"
read name
echo "Please enter your age"
read age
echo "Please enter your sex. Male/Female"
read sex
echo "So you're a $age year old $sex called $name"
```

read allows the user to input information where it is then stored in the variable defined after the read. *read variable* would take whatever input the user entered and store it in $variable. We then access this with echo and set up a neat sentence. This script is reasonably messy though; read has another function that could halve the size of this script.

```
clear
read -p "Please enter your name  : " name
read -p "Please enter your age  : " age
read -p "Please enter your sex. Male/Female  : " sex
echo "So you're a $age year old $sex called $name"
```

That is more efficient code. However it's still a bit messy when run. A solution? Good old white spaces!

```
clear
read -p "Please enter your name  : " name
echo ""
read -p "Please enter your age  : " age
echo ""
read -p "Please enter your sex. Male/Female  : " sex
echo ""
echo "So you're a $age year old $sex called $name"
```

Now we have an efficient and clean Bash script.

If Statements

An if statement can be used to check for something and do something else depending on the outcome of the check. For example, if I had an 'apple', I would want to make sure it's still an 'apple' and not an 'orange' because I don't like Oranges!

The syntax for an if statement is

```
if [something]
        then
        elif
```

```
        then
        elif
        then
        ....etc....
        else
fi
```

The else if statement or (elif) is not necessary, but it can be used if needed.

An if statement to check if our $fruit variable is an 'apple' would look like this

```
echo "Please enter type of fruit"
read fruit

if [ $fruit = apple ]
        then echo "Good, I like Apples"
        else echo "Oh no, I hate Oranges!"
fi
```

Just to explain this statement,

```
if [ the contents of $fruit is 'apple' ]
        then say "Good, I like Apples"
        if it's not, then say "Oh no, I hate Oranges!"
finish
```

If statements are an easy concept to grasp as they are similar to the "if" used in spoken English. But say you wanted to have 4 or 5 checks, the answer may be to write 4 or 5 statements but that's not the most practical way. This is where elif comes in handy.

```
if [ $fruit = apple ]
        then echo "Good, I like Apples"
elif [ $fruit = pear ]
        then echo "Good, I like Pears"
elif [ $fruit = banana ]
        then echo "Good, I like Bananas"
        else echo "Oh no, I hate Oranges!"
fi
```

This saves us from from repetitive scripting. There are better ways to check what the fruit is, but we won't go into that now.

Storing application stdout to a variable:

Application stdout (what you see on the terminal screen, with an un-piped application) can be saved and used in Bash. The simplest and most elegant way is to use command substitution, by wrapping the code in $(...)

Example

```
fooVar=$(who)
echo $fooVar
```

This code should output the current users, their respective ttys, and date of login. Note that this strips newlines. Be sure to do any parsing in line (| grep, etc) and then pass it to a variable. We will try this again, but grep for tty7, the GUI's tty.

Example 2

```
fooVar=$(who | grep tty7)
echo $fooVar
```

This should output the single user that is currently logged into the WM. Let's move on to more advanced data manipulation within command substitution.

FUNctions

Bash lets you create a function on the fly, really handy if you plan on using a code block more then once. Functions reduce the amounts of editing you have to do in a script, if and when you have to update your script. Let's get to it!

Example

Here is an example script:

```
echo "echo is Called"
echo "Functions are FUN!"
```

```
echo "echo is Called"
```

Although this example is simple, you can see that if you want to change "echo is Called" to say "foo is Called" you would have to edit twice.

Below is the same app using functions.

```
echoFunction() {
  echo "echo is Called"
}
fooBar() {
  echo "Functions are FUN!"
}

echoFunction;
fooBar;
echoFunction;
# You call functions without (), just the function name then a semicolon.
```

This example, as you can see may be longer now, but you can imagine how, adding features, this will eliminate code and reduce complexity. Also, you can see if you want to change the echo call, you have to edit one line, not two.

Debugging

I always find it useful to trace a script to find out why something does not work as expected. To trace, start it through bash explicitly and use the -x option, like so:

```
bash -x ./script.sh
```

This will write each command to standard error (preceded by a '+ ') before it is executed.

Other Scripting Languages related to Bash

tr

tr is one of the most basic applications to pipe data through that uses a basic scripting syntax. In this case, it accepts Regular Expressions. Let's do a normally complicated task, transforming a string to all uppercase.

Example

```
read foo
var=$(echo $foo | tr "{a-z}" "{A-Z}")
 # {a-z} Matches a through z
 # {A-Z} matches A through Z
echo $var
```

The output should look something like this:

```
this is a test
THIS IS A TEST
```

tr also can TRanslate strings, so let's translate all "tar" in $foo to "bar".

Example

```
echo "Type in: I love tars"
read foo
var=$(echo $foo | tr "t" "b")
echo $var
```

the output should look something like this:

```
I love tars
I love bars
```

AWK

AWK (Short for Aho, Weinberger & Kernighan)

awk has its own custom scripting language, suitable for a tutorial by itself, so I will cover only the basics to help assist when you are bash scripting. This is not meant to be complete or comprehensive in any way.

pidof clone

Let's make a quick pidof clone that prompts for a process identifier, then echoes the process ID.

```
read pname
ps -ef | grep -v grep | grep $pname | awk '{print $2}'
```

Let's take some pipes out and use only awk for the filtering

```
read pname
ps -ef | awk -v p=${pname} '$8 ~ p { print $2 }'
```

Single quotes are used to pass the *awk* command(s). The curly braces are to use the *awk* language (for stuff like prints, ifs, etc.). *Print* prints the column passed given by the $ markup, space delimited.

The awk -v option allow passing a shell value into an awk variable, the $8 is a field variable (column 8 of the ps -ef command's output) and the operator ~ is a regular expression match.

There are a lot more commands than the print command, including if statements, etc., and is worth looking into if you are interested in what you see here!

SED

sed is one of the most complicated scripting languages on the GNU / Linux system. I am only going to cover the s/ command here.

Basic Substitution

Try this out to show that *sed* can not only replace inline, but unlike *tr*, replace with a longer or shorter string than before.

```
read foo
echo $foo | sed "s/foo/bars/"
```

When this command is run, it should substitute the first appearance of "foo" with "bars".

This is an example of the output.

```
I love to go to foo
I love to go to bars
```

CategoryCommandLine

Beginners/BashScripting (last edited 2013-12-19 18:17:59 by larsnooden @ dsl-roibrasgw1-50de5f-109.dhcp.inet.fi[80.222.95.109]:larsnooden)