Bash

Cheatsheet

P J

Contents

1 Bash Cheatsheet

A Bash script is a plain text file which contains a series of commands.

These commands are a mixture of commands we would normally type ouselves on the command line (such as `ls` or `cp` for example).

> Anything you can run normally on the command line can be put into a script and it will do exactly the same thing. Similarly, anything you can put into a script can also be run normally on the command line and it will do exactly the same thing.

You don't need to change anything. Just type the commands as you would normally and they will behave as they would normally. It's just that instead of typing them at the command line we are now entering them into a plain text file. In this sense, if you know how to do stuff at the command line then you already know a fair bit in terms of Bash scripting.

It is convention to give files that are Bash scripts an extension of `.sh`.

As you would be aware, Linux is an extensionless system so a script doesn't necessarily have to have this characteristic in order to work.

1.1 How do they work?

This is just a little bit of background knowledge. It's not necessary to understand this in order to write scripts but it can be useful to know once you start getting into more complex scripts (and scripts that call and rely on other scripts once you start getting really fancy).

In the realm of Linux (and computers in general) we have the concept of programs and processes. A program is a blob of binary data consisting of a series of instructions for the CPU and possibly other resources (images, sound files and such) organised into a package and typically stored on your hard disk. When we say we are running a program we are not really running the program but a copy of it which is called a process. What we do is copy those instructions and resources from the hard disk into working memory (or RAM). We also allocate a bit of space in RAM for the process to store variables (to hold temporary working data) and a few flags to allow the operating system (OS) to manage and track the process during it's execution.

2 Variables

Variables are one of those things that are actually quite easy to use but are also quite easy to get yourself into trouble with if you don't properly understand how they work. As such there is a bit of reading in this section but if you take the time to go through and understand it you will be thankful you did later on when we start dabbling in more complex scripts.

2.1 Define Variables

Variables may have their value set in a few different ways. The most common are to set the value directly and for its value to be set as the result of processing by a command or program.

```
#!/bin/bash

VarName=$(ls /tmp)
variable_name="Data List"
echo $VarName
echo $variable_name
```

Note there is no space on either side of the equals (=) sign.

Variable names may be uppercase or lowercase or a mixture of both but Bash is a case sensitive environment so whenever you refer to a variable you must be consistent in your use of uppercase and lowercase letters. You should always make sure variable names are descriptive. This makes their purpose easier for you to remember.

To read the variable we then place its name (preceded by a `$` sign) anywhere in the script we would like. Before Bash interprets (or runs) 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.

2.2 Command line Arguments

When we run a program on the command line you would be familiar with supplying arguments after it to control its behaviour. For instance we could run the command `ls -l /etc`. `-l` and `/etc` are both command line arguments to the command `ls`. We can do similar with our bash scripts. To do this we use the variables `$1` to represent the first command line argument, `$2` to represent the second command line argument and so on. These are automatically set by the system when we run our script so all we need to do is refer to them.

```
#!/bin/bash
# A simple copy script
cp $1 $2
# Let's verify the copy worked
echo Details for $2
ls -lh $2
```

2.3 Other Special Variables

- `$0` - The name of the Bash script.
- `$1`, `$2`, `$3`, etc.. - The first, second, third, etc.. argument of the script.
- `$#` - How many arguments were passed to the 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.
- `$USER` - Username of the user running the script.
- `$HOSTNAME` - Hostname of the machine the script is running on.
- `$SECONDS` - Number of seconds since the script was started.
- `$RANDOM` - Returns a different random number each time is it referred to.
- `$LINENO` - Returns the current line number in the Bash script.

Some of these variables may seem useful to you now. Others may not. As we progress to more complex scripts in later sections you will see examples of how they can be useful.

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

Here is a simple example to illustrate their usage:

```
#!/bin/bash

# A simple variable example
myvariable=Hello
anothervar=Fred
echo $myvariable $anothervar
echo
sampledir=/etc
ls $sampledir
```

```
$ ./simplevariables.sh
Hello Fred
a2ps.cfg aliases alsa.d ...
$_
```

> Variables can be useful for making our scripts easier to manage. Maybe our script is going to run several commands, several of which will refer to a particular directory. Rather than type that directory out each time we can set it once in a variable then refer to that variable. Then if the required directory changes in the future we only need to update one variable rather than every instance within the script.

2.4 Quotes

In the example above we kept things nice and simple. The variables only had to store a single word. When we want variables to store more complex values however, we need to make use of quotes. This is because under normal

circumstances Bash uses a space to determine separate items.

```
$ myvar=Hello World
-bash: World: command not found
$_
```

Remember, commands work exactly the same on the command line as they do within a script.

> Because commands work exactly the same on the command line as in a script it can sometimes be easier to experiment on the command line.

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).

```
myvar='Hello World'
echo $myvar # -> Hello World
newvar="More $myvar"
echo $newvar # -> More Hello World
newvar='More $myvar'
echo $newvar # -> More $myvar
```

2.5 Command Substitution

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 practices, preceded by a $ sign.

```
myvar=$( ls /etc | wc -l )
echo There are $myvar entries in the directory /etc
# -> There are 126 entries in the directory /etc
```

```
$ ls
bin Documents Desktop ...
Downloads public_html ...
$ myvar=$( ls )
$ echo $myvar
bin Documents Desktop Downloads public_html ...
$_
```

> When playing about with command substitution it's a good idea to test your output rather than just assuming it will behave in a certain way. The easiest way to do that is simply to `echo` the variable and see what has happened. (You can then remove the echo command once you are happy.)

2.6 Exporting Variables

Remember how in the previous section we talked about scripts being run in their own process? This introduces a phenomenon known as scope which affects variables amongst other things. The idea is that variables are limited to the process they were created in. Normaly this isn't an issue but sometimes, for instance, a script may run another script as one of its commands. If we want the variable to be available to the second script then we need to export the variable.

- script1.sh

```
#!/bin/bash
# demonstrate variable scope 1.
var1=blah
var2=foo
# Let's verify their current value
echo $0 :: var1 : $var1, var2 : $var2
export var1
./script2.sh
# Let's see what they are now
echo $0 :: var1 : $var1, var2 : $var2
```

- script2.sh

```
#!/bin/bash
# demonstrate variable scope 2
# Let's verify their current value
echo $0 :: var1 : $var1, var2 : $var2
# Let's change their values
var1=flop
var2=bleh
```

Now lets run them.

```
$ ./script1.sh
script1.sh :: var1 : blah, var2 : foo
script2.sh :: var1 : blah, var2 :
script1.sh :: var1 : blah, var2 : foo
$_
```

The output above may seem unexpected. What actually happens when we export a variable is that we are telling Bash that every time a new process is created (to run another script or such) then make a copy of the variable and hand it over to the new process. So although the variables will have the same name they exist in separate processes and so are unrelated to each other.

Exporting variables is a one way process. The original process may pass variables over to the new process but anything that process does with the copy of the variables has no impact on the original variables.

Exporting variables is something you probably won't need to worry about for most Bash scripts you'll create. Some-

times you may wish to break a particular task down into several separate scripts however to make it easier to manage or to allow for reusability (which is always good).

3 User Input

We looked at one form of user input Command line Argumets in the previous section. Now we would like to introduce other ways the user may provide input to the Bash script.

3.1 Ask the User for Input

If we would like to ask the user for input then we use a command called `read`. This command takes the input and will save it into a variable.

```
1 #!/bin/bash
2
3 read Variable_Name
4 echo $Variable_Name
```