

# An Introduction to Bash Scripting

## Solomon Chang

### Comments

All comments begin with the # symbol.

However, at the beginning of every Bash script is a special comment:

```
#!/bin/bash
```

This comment tells the script what should parse this script. Here is a script that does nothing:

```
#!/bin/bash
# I do nothing
```

Okay, let's do something, then

The following script will output *Good Morning*:

```
#!/bin/bash
echo "Good Morning"
```

(The quotes keep a string whole.) You will probably use the echo command a lot to output debug checkpoints when you are writing a Bash script.

### Setting and Reading Variables

You can create a variable out of the blue as follows:

```
myvar="Good Morning"
```

You can echo that variable with the following statement:

```
echo $myvar
```

When setting a Bash variable, there are two counter-intuitive points to remember:

- Spacing is important. Notice that there are no spaces before or
- Do not use a dollar sign (\$). You use a dollar sign when you are reading the variable, but not when you are setting it.

That having been said, you can use variables from within quotes:

```
myname="Solomon"
echo "Good morning, $myname"
# outputs Good Morning, Solomon
```

But if you need to literally output the dollar sign, use the backslash character:

```
myname="Solomon"
echo "Good morning, \$myname"
# outputs Good Morning, $myname
```

### Casting to a Number

For the most part, bash variables are almost always strings. Every now and then, you will need to evaluate a Bash variable as a number. To do so, use the  $$(())$  operator:

```
x=$((4))
# $x is now 4, the integer
```

You can also perform some basic mathematic operations with the  $$(())$ ...

```
x=$((4))
echo $((x + 1))
# prints "5"
```

### Piping

The output of a command can be "piped" into other commands using the pipe character (`|`). The following

command takes a directory listing, and returns only the last line:

```
ls -l | tail -n 1
# returns details of the last file of the current directory
```

The contents of the command "ls -l" is above is used a standard input to the "tail -n 1" command, thus returning the last file of the current directory.

You can use the pipe character multiple times. You could take the output from the above command and output it to *another* pipe. For example:

```
ls -l | tail -n 1 | wc -c
# returns the number of letters contained in the last line of a wc -l command.
```

## Redirection

If you do not want the output to be returned to standard out, you can redirect the output instead to a file using the > and >> indicators:

```
ls -l | tail -n 1 | wc -c > /tmp/num_of_letters.txt
# creates a new file, /tmp/num_of_letters.txt, with some integer.
```

```
ls -l | tail -n 1 | wc -c >> /tmp/num_of_letters.txt
# appends to the end of the file, /tmp/num_of_letters.txt, with some integer.
```

This is extremely useful for grabbing output of some kind of status commands and storing them in some kind of file.

## Literals

Backticks (`) can be used to indicate literals. That is, the output of the command contained in the backticks can be used for the standard input of some other command, or you can even set it to a variable like so:

```
v=`ls -l | tail -n 1 | wc -c`
# $v is now some string representing a number.
```

One can also cast the above example into an integer:

```
v=$((`ls -l | tail -n 1 | wc -c`))
# $v is now some integer.
```

## Expressions

Expressions are enclosed in square brackets and are used in *if* and *while* statements. They return true if the condition evaluates to true and false if the expression evaluates to false. A few examples of expressions are as follows:

```
[ $x = "five" ]           #True if $x equals "five" (string comparison)
[ $x -eq 5 ]              #True if $x equals 5
[ $x -ne 5 ]              #True if $x does not equal 5
[ ! $x -eq 5 ]            #True if $x does not equal 5
[ $x -gt 5 ]              #True if $x is greater than 5
[ $x -lt 5 ]              #True if $x is less than 5
[ -f /var/log/mysql.log ] #True if /var/log/mysql.log exists and is a regular file
[ -S /var/run/mysql.sock ] #True if /var/run/mysql.sock exists and is a socket file
```

Spacing is extremely important! The first bracket must be followed by a space, and the last bracket must be preceded by a space, or you will get a syntax error!

There are many more expressions for Bash, easily found through online resources.

## If...Then...[Else...]Fi

The if statement evaluates an expression, and if it is true, executes code located between the following *then* keyword to the outmost *fi* keyword. (I say outmost because if statements could be nested within each other.) Alternatively, you could also have an *else* clause before the *fi* to indicate a block of code to execute if the

expression does not evaluate to true. Here are a few examples:

```
if [ $(`date +%H`) -lt 12 ]
then
echo "Good Morning"
fi
```

Caveat: some coders use semicolons instead of line breaks in their code. I'm one of them, so my code looks more like:

```
if [ $(`date +%H`) -lt 12 ]; then
    echo "Good Morning"
fi
# I also use indentation for clarity.
```

Including an else statement would look like:

```
if [ $(`date +%H`) -lt 12 ]; then
    echo "Good Morning"
else
    echo "Good Day"
fi
```

Any of those code blocks can include further nested *if* statements:

```
if [ $(`date +%H`) -lt 12 ]; then
    echo "Good Morning"
else
    if [ $(`date +%H`) -lt 17 ]; then
        echo "Good Afternoon"
    else
        echo "Good Evening"
    fi
fi
```

Just remember to end all your *if* blocks with a *fi*.

## Loops

A loop is a programming structure where a block of code executes for a certain number of repetitions, and certain variables may change for each iteration of the loop.

There are two different commands used for loops:

- for
- while

A for loop is useful for iterating through some kind of list. A while loop is useful for iterating until some specific condition is no longer true.

### for

The *for* command will take a list and perform an iteration for each item on that list:

```
for x in "hello" "world"; do
    echo $x
done
# prints "hello" and "world" on two different lines
```

(Once again, note that the initial assignment of *x* does not include a dollar sign, but reading the variable does.

More often the case, however, is to have the literal output of some command to supply the list of items for iteration:

```
for x in `ls /var/backup/mail_*`; do
    rsync -U $x 10.0.0.33:/var/backup/remote
done
# rsyncs all files in /var/backup beginning with mail to another server
```

I often use something similar to backup a list of specifically important tables in a database:

```
for x in `mysql test -e "show tables like 'archive_%'`; do
  mysqldump test $x > /var/backup/archive_`date +%Y%m%D`
done
# for every table starting with "archive" create a new file that includes date.
```

## while

The *while* command will loop everything between *do* and *done* until the expression is no longer true. This has more potential for infinite loops if you do not account for an exit condition.

```
x=5
while [ x -gt 1 ]; do
  echo $x
  x=$((x - 1))
done
# prints 5, 4, 3, 2 each on a different line
```

However, sometimes, an infinite loop can be useful:

```
while [ 1 -eq 1 ]; do
  if [ `$(ping -c 1 yahoo.com | wc -l)` -gt 0 ]; then
    echo "I can see Yahoo!"
  else
    echo "The world is coming to an end!"
  fi
done
```

## Going to Background

If you're running an aforementioned infinite loop, you can run it in the background by adding an ampersand to your command when you call the script. Let's say you created a script called *test.sh*. You would execute it in the background with the command:

```
sh test.sh &
```

If it is running in the foreground, you can also halt it with CTRL-Z, followed by the *bg* command to send it to the background.

To bring it back to the foreground, use the *fg* command.

Note that it is always quite possible that your script finished executing, or crashed, while running in the background and you do not notice. If your script seems to have disappeared, this is usually the case.