Introduction to Bash Scripting

by Zane Beckwith (CAP REU Grad Assistant)

The window you've been entering commands into as you learn Linux/Unix is called a “shell”. As you can probably tell already, it's the standard way of doing things in Linux/Unix. The primary reason for this is that the commands you can enter into the shell actually constitute a surprisingly-powerful programming language, one that allows you to automate many repetitive tasks and glue together the different parts/programs of your project.

For example, say you've got a bunch (maybe hundreds) of data files in a directory; these could be from astronomical observations, or results from simulation code, or whatever. Say you've also written a Python script that analyzes the data somehow (I don't care about what it does for right now). You want to grab all of the data with some particular value for the parameter “field”, throughout all the hundreds of files, run these data through the Python script, and save that output file in some directory with a convenient name.

Easy, right? Well, remember, you potentially have hundreds of data files; do you really want to sit and do that for each one of them? And what about tomorrow, when you want to look at a different field value? Or worse (but not at all uncommon): you find the Python script had a mistake in it, and now you have to re-run all that analysis. What you need is to write a script to do this work for you.

What Is a Shell Script?

When performing simple tasks, you can enter commands into the shell individually, as you've been doing so far. However, for more extensive jobs, you'll want to collect a string of commands together into a single file, so they can all be run with one simple command. This is a shell script. Conventionally, a shell script has the file extension “.sh”.

There are, in fact, a bunch of different shell programs you can choose from. The one you're using right now is called the tc-shell, or tcsh. However, it's kind of a limited shell, so the one we'll be using is the bash shell (it's one of the most popular shell programs). If you want to interactively try some of the things discussed here, you need to type bash into your shell; this switches your shell from tcsh to bash. When you type exit, you exit the bash program and return to using tcsh.

However, you can write a script to be used by the bash shell, and it won't matter what shell the user of your script is in fact using. This brings us to **step one in creating a shell script.** To tell the system that your script should be run using bash, start the file with the following line (frequently called a “sha-bang”):

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

This is very important, because the syntax of different shells can vary greatly. What all of this means is that, when you run a script, the shell you're in actually starts a separate bash shell in the background to run your script, then closes it and returns to your shell when the script is finished.

Now for **step two in making a shell script.** To make the script file an “executable” file, you must give yourself the proper access to it, using the chmod command. If our script file is called “myScript.sh”, the command is:

```
chmod u+x myScript.sh
```

This means “CHange the MODe of the file so that the User ADDs eXecutable privileges.”

Exercise: Use ls -l before and after doing the chmod to see how the permissions flags change.

Finally, the **third step in making a shell script** is to actually write the script; that's what the rest of these notes will introduce you to.

To make the requisite “Hello world” program, for example, we need one more, very useful, command: echo. This command just prints whatever argument you give it.

So, here's our simple hello world script:

```
#!/bin/bash
echo “Hello, world!”
```

We would save that file as helloWorld.sh, run chmod u+x helloWorld.sh in the shell, then when we run ./helloWorld.sh, the shell prints “Hello, world!”.

Exercise: Try it.

One last comment (pun very much intended): to write a comment in your shell script, precede it with the hash-sign (#). Thus the line

```
echo “Hello, world!”        # here I say hello
```

prints “Hello, world!”, but the part after the hash is a comment telling the reader of your script file what that line does.

Variables

As I said, the shell is really a programming language, which means you need variables. You define a variable like so

```
X="hello"
```

and refer to it as

```
$X
```

Remember, when you run a shell script, the shell you're in actually opens a different shell to run your script; this means the variables you defined in the script are no longer defined in your shell once the script has finished running.

Here are a few tricky points about assigning variables:

1) Make sure not to leave space on either side of the = sign. For example, the following gives an error message:

```
X = 5
```

It should be

```
X=5
```

2) Variables can be numbers (like 5), or "strings" (like hello). Surrounding a word with quotation marks indicates that it should be a string. However, using quotation marks is not always necessary (the shell can easily tell that hello is a word while 5 is not); where you really need them is when your variable includes spaces. For example

```
X=hello                 # OK
X='hello'               # OK
X=hello world           # error
X='hello world'         # OK
```

This, and the problem with using spaces mentioned in #1, is because the shell essentially sees your script as a bunch of commands and command arguments, separated by spaces. For example,

```
foo=bar
```

is considered a command. The problem with

```
foo = bar
```

is that the shell sees the word foo by itself and interprets it as a command, which it's not. Likewise, the problem with

```
X=hello world
```

is that the shell interprets X=hello as a command, the assignment command, and the free-standing word world does not make any sense (since X=hello can't take arguments).

3) Single-quotes and double-quotes mean slightly different things in bash scripting. Single quotes, like

```
X='i need some $money'
```

mean to use that exact phrase, so

```
echo $X
```

would print out i need some $money. On the other hand double quotes, like

```
Y="i need some $money"
```

will expand variables. So if $money='cash', then

```
echo $Y
```

will print out "i need some cash".

4) If we need to use double quotes, but also need to print a dollar-sign, $, we need to "escape" it (print it literally), by putting a backslash in front of it. So, if $needs=5, then

```
Z="i need \$ $needs"
echo $Z
```

would print out "i need $ 5".

5) In general, it's a good habit to surround any string variable with quotation marks (so, use x='hello', rather than x=hello). Further, it's also a good habit to use single quotes unless you need to use double quotes.

6) Suppose you want to echo the value of the variable X, followed immediately by the letters "abc". How do you do this? If we try

```
X=ABC
echo "$Xabc"
```

we get no output. What went wrong? The answer is that the shell thought we were asking for the variable Xabc, which we haven't set and is thus empty. The way to deal with this is to put braces around X to separate it from the other characters. The following gives the desired result:

```
X=ABC
echo "${X}abc"
```

Exercise 1: Define two variables, say $ten and $one. The first variable holds the first digit in the price of some product and the second holds the last digit (eg., if the price is $59 then $ten=5 and $one=9). Now, report the full price of the item (in the above example, you would print out "$59"). Yes, I know this is a silly example, but imagine you didn't put those variables in by hand but rather they were given to you from another program or an earlier part of the script, and you need to put them together this way.

Note: my solutions to these exercises are in the "solutions" directory.

Command Line Arguments

Frequently, you'll want to pass arguments to your scripts, the way we pass the name of the directory we're interested in to the command ls, for example. To do this, you use the bash environment variables $1, $2, $3, etc., where $1 is the first argument, $2 is the second, and so on.

Take, for example, a script file called argTest.sh:

```
#!/bin/bash
echo “$1”
```

When this is run like ./argTest.sh hello, the output will be “hello”.

The variable $0 is, somewhat surprisingly, the name of the script itself. This can be useful so that the program knows what its own name is. For example, if the user calls ./foo.sh firstArg secondArg, then $0=”./foo.sh”.

Another useful environment variable is $@, which expands to ALL of the command line arguments. In the example just given, $@=”firstArg secondArg”.

Lastly, $# expands to the NUMBER of arguments ($# = 2 in the example above). Note that the variable $0 is not included in the number $#, so in this example the value of $# would be 2 and not 3.

Exercise 2: Write a script that takes your name as an argument, and prints Hello <yourname>!

Conditionals (if)

Sometimes, it's necessary to check that certain conditions are true or false. For example: does a string have 0 length; does the file "foo" exist? We use the if command to run a test. The syntax is as follows:

```
if condition
then
    statement1
    statement2
    ..........
fi
```

If the command “condition” returns “0”, the statements between then and fi are executed.

Sometimes, you may wish to specify an alternate action when the condition fails. Here's how it's done:

```
if condition
then
    statement1
    statement2
    ..........
else
    statement3
    ..........
fi
```

Alternatively, it is possible to test for another condition if the first if fails.

```
if condition1
then
    statement1
    statement2
    ..........
elif condition2
then
    statement3
    statement4
    ........
elif condition3
then
    statement5
    statement6
    ........
fi
```

Note that any number of elifs can be added.

Any command can go in place of the “conditions”, and the block will be executed if and only if the command returns an exit status of 0 (in other words, if the command “exits successfully”).

For example,

```
if ls *.txt
```

will only evaluate to true if ls returns 0, which only happens if there are .txt files in the current directory. However, the most common command to use as a test condition is the test or [] command.

The Test Command and Operators

The command used in conditionals nearly all the time is the test command. Test returns true or false (more accurately, exits with 0 or non-zero status) depending respectively on whether the test is passed or failed. It works like this:

```
test operand1 operator operand2
```

For some tests, there need be only one operand (operand2)

The test command is usually abbreviated as

```
[ operand1 operator operand2 ]
```

So, for example, let me introduce the numerical equality operator: -eq (see below for a summary of some useful operators). If we have $a=4 and $b=5 then

```
[ $a -eq $b ]
```

will evaluate to 1 (false).

To see the result of a test, such as the one above, when using the shell interactively, enter the command echo $? after the test command. The shell variable $? holds the exit value of the last command used. Remember: “0” means “true” and anything other than zero means “false”.

To bring this discussion back down to earth, here's an example:

```
X=3
Y=4
if [ $X -lt $Y ]       # -lt tests if operand1 < operand2
then
    echo "\$X=$X, which is smaller than \$Y=$Y"
fi
```

A Few More Points About Tests

1) Because it can be annoying to have to write the if and the then on separate lines, you'll frequently see them put on the same line. In general, commands are separated by putting them on separate lines, but to put multiple commands on one line, you just separate them with a semi-colon. So the if/then/fi will frequently look like

```
if [test]; then

statements

.....

fi
```

2) Numbers and strings have two different sets of test operators. I give below a very short list of some of the basics; searching the internet for “bash test operators” or something like that is useful to find out all the different

operators.

3) Spaces are very important with if blocks and test statements. For one thing, there must be a space between if and the first bracket. Also, within the test, there must be a space between each bracket, each operand, and the operator. So for example,

```
if[ $a -eq $b ]
```

would be a fatal error, because the shell doesn't know what the if[command is, it only knows if. Likewise

```
if [ 1-eq2 ]
```

would always evaluate to true. Why? Because if the test operator gets only one string between the brackets, it automatically evaluates to true, and 1-eq2 is just one string, not three. This can be very frustrating, and doesn't make itself known, so watch out.

4) It's generally a good idea to enclose variables that are strings inside quotations marks when referencing them. What I mean is that, if you write

```
x='hello world'
y='hello world'
if [ $x = $y ]
```

the script will cause an error (“too many arguments”). What happened? Again, strings with spaces cause a problem: the shell sees [hello world = hello world]. That's one operator and FOUR operands. To avoid this, make it a habit to enclose variable references, when the variables are strings, in quotation marks:

```
x='hello world'
y='hello world'
```

```
if [ “$x” = “$y” ]
```

(remember to use double-quotes, or else those will just become $x and $y, not 'hello world'.

5) A test can be negated by putting an exclamation mark in front of it. For example

```
! [ 1 -eq 2 ]
```

evaluates to true.

6) A useful command to learn here is “exit”, which basically does what it says. If your program finds a problem somewhere, you can stop it and return an exit code to the user. Here's an example:

```
status=”bad”
if [ “$status” = “bad” ]
then
    echo “Something bad has happened!”
    exit 1
fi
```

By giving the exit command the argument 1, we return the exit value 1 to the shell that called the script; conventionally, an exit value of 1 indicates an error.

String Comparison Operators

=	equal to
!=	not equal to
>	greater than (ASCII alphabetical order)
<	less than (ASCII alphabetical order)
-z	string is null (zero length)
-n	string is NOT null

Numerical Comparison Operators

-eq	equal to
-ne	not equal to

-gt	greater than
-lt	less than
-le	less than or equal to
-ge	greater than or equal to

Exercise 3: Re-write the script from exercise 2, but now check that the user has in fact given their name as a command line argument. If they haven't, exit the program and inform them of the proper usage.

Loops

Loops enable you to reiterate a procedure or perform the same procedure on several different items.

For loops

The syntax for the “for” loop is best demonstrated by an example:

```
for X in red green blue
do
    echo $X
done
```

The for loop iterates the loop over the space-separated items. Note that if some of the items have embedded spaces, as always you need to protect them with quotes. Here's an example:

```
colour1="red"
colour2="light blue"
colour3="dark green"
for X in "$colour1" $colour2" $colour3"
do
    echo $X
    done
```

Exercise 4: Write a script that goes through all the files in a directory (given in the first command line argument, otherwise the current directory) and prepends the word “test” to the beginning of any that have “.dat” suffixes. You can create some dummy .dat files to play with, and clearly you don’t want to play with this script in a directory that actually has important .dat files in it! Hint: you’ll find the wildcard * useful here.

While Loops

While loops iterate "while" a given condition is true. Here's an example:

```
    X=0
    while [ $X -le 20 ]; do
        echo $X
        X=$(expr $X + 1)      # this will be explained in a minute (“command
    substitution”)
    done
```

Exercise 5: Write a script that asks you for a file name, then tells you whether or not that file is in the current directory. Typing “stop” should end the program. For this, you should look into the “read” command, which takes input from the user at the command line.

Command Substitution

Command substitution enables you to take the output of a command and use it for something else (e.g. assign it to a variable). Command substitution works as follows: $(commands) expands to the output of “commands”.

Here's an example:

```
    files="$(ls)"
```

```
echo "$files"                 # a list of the files in the current directory
X=$(expr 3 \* 2 + 4)
echo $X                       # expr evaluates arithmetic expressions
```

Final exercise:

Let's do that example I gave at the beginning of this scripting tutorial. In the “final” directory there's a sub-directory called “data”, with a few data files in it. Also, there's a Python script called “analyze.py”, which can analyze the data in these files.

This Python script takes two command line arguments: the first is the name of the .dat file to be analyzed, and the second is what you want the output file to be named. Careful: “analyze.py” should only be used to analyze data files with the extension “.dat”; other data files contain a different type of data. You may want to take a look at the .dat files to see how the data are stored.

Imagine that, rather than 3 data files, there are hundreds, and you want to analyze the data in all of these files as long as the entry “field=” has a particular value. Write a script that takes the field value as a command line argument (eg., calling “myScript.sh 3” will analyze only lines with “field=3”), saves only rows of data that have that field value (hint: “grep” is a good tool for such things), runs all of this data through the Python script, then saves the resulting output to a .out file in a directory named with today's day of the week.

My solution script is in the “final” directory, and it's called “solution.sh”.