Introduction to Bash Scripting

The window you've been entering commands into as you learn Linux/Unix is called a “shell”. As you can probably tell so far, it is 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 very 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. What's important to me is that the data is organized into columns: the first is an x-value, then a y-value, then a column of “field” values (recorded as “field=2” for example), then maybe some other things. Say you've also written a Python script that analyzes that 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 “field”, run them 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”. 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. The sha-bang indicates that the script should be run using the program /bin/bash, which is the bash shell, regardless of which interactive shell the user is currently running; specifically, 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.

To make the shell allow you to run your script file as a program, 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

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

Just like when typing interactively in the shell, commands are put on separate lines; to put multiple commands on one line, just separate them with a semi-colon.

To make the requisite “Hello world” program, 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!”.

**Try it.**

Using the Right Shell

As I said above, there a few different shell programs. The one you're using right now is called the tc-shell, or tcsh. However, it's kind of a simplistic shell, so the one we'll be using is the bash shell.

Again, no matter the shell you're currently in, if you include #!/bin/bash at the top of your script file, it will run fine. However, 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.

### Variables

Remember, the shell is really a programming language. Any programming language needs variables. You define a variable as follows:

X="hello"

and refer to it as follows:

$X

Some things to take note of regarding semantics:

1) Bash gets unhappy if you leave a space on either side of the = sign. For example, the following gives an error message:

X = hello

It should be

X=hello

2) While quotes are used in the example above, they are not always necessary. Where you need quotes is when your variable names include spaces. For example

X=hello world # error  
 X="hello world" # OK

This, and the problem with “X = hello”, is because the shell essentially sees the command line as a pile of commands and command arguments seperated by spaces. For example, “foo=bar” is considered a command. The problem with “foo = bar” is that the shell sees the word “foo” seperated by spaces and interprets it as a command, which it's not. Likewise, the problem with the command “X=hello world” is that the shell interprets “X=hello” as a command, and the free-standing word "world" does not make any sense (since the assignment command doesn't take arguments).

Remember, when you run a shell script, the shell you're in actually opens a different shell to run your script; specifically, once the script has finished running, the variables you defined in it aren't defined any longer.

#### Using Braces to Protect Your Variables

OK, here's a potential problem situation: suppose you want to echo the value of the variable X, followed immediately by the letters "abc". How do you do this? Let's have a try :

#!/bin/bash

X=ABC

echo "$Xabc"

This gives no output. What went wrong? The answer is that the shell thought we were asking for the variable Xabc, which is uninitialised. The way to deal with this is to put braces around X to seperate it from the other characters. The following gives the desired result:

#!/bin/bash

X=ABC

echo "${X}abc"

**Exercise 1: Create a script that stores the variable x=5, then prints out “The value of $x is 5”.**

**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/then/elif

Sometimes, it's necessary to check for certain conditions. For example: does a string have 0 length; does the file "foo" exist? We use the “if” command (ended by the “fi” 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. Note that any number of “elifs” can be added.

if condition1

then

statement1

statement2

..........

elif condition2

then

statement3

statement4

........

elif condition3

then

statement5

statement6

........

fi

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 direcotry. 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 typically abbreviated in this form:

[ operand1 operator operand2 ]

So, for example, if we have a=4 and b=5

[ $a -eq $b ] # -eq tests for numerical equality

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:

#!/bin/bash

X=3

Y=4

empty\_string=""

if [ $X -lt $Y ] # is $X less than $Y ?

then

echo "\$X=${X}, which is smaller than \$Y=${Y}" # \$x prints “$x” rather than the variable x

fi

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

#!/bin/bash

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.

#### Some Pitfalls To Look Out For

The test command needs to be in the form "operand1<space>operator<space>operand2" or operator<space>operand2 ; in other words, you *really* need these spaces, since the shell considers the first block containing no spaces to be either an operator (if it begins with a '-') or an operand (if it doesn't). So for example,

if [ 1-eq2 ]; then

echo "hello"

fi

gives exactly the "wrong" output (i.e. it echoes "hello"), since it sees an operand but no operator.

Another potential trap comes from not protecting variables in quotes. There are a lot of good reasons for using quotes all the time, or almost all of the time. Failing to do this when you have variables expanded inside tests can result in very weird bugs. Here's an example:

#!/bin/bash

X="-n"

Y=""

if [ $X = $Y ] ; then

echo "X=Y"

fi

This will give misleading output since the shell expands our expression to

[ -n = ]

and the string "=" has non zero length.

Searching the internet for “bash test operators” or something like that is useful to find out all the different operators used with the “test” or “[ ]” command.

**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 are constructions that enable one 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 example.

#!/bin/bash

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, you need to protect them with quotes. Here's an example:

#!/bin/bash

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 adds 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!**

#### While Loops

While loops iterate "while" a given condition is true. An example of this:

#!/bin/bash

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 is a very handy feature of the bash shell. It enables you to take the output of a command and use it for something else (e.g. assign it to a variable).

There are two means of command substitution: brace expansion and backtick expansion.

Brace expansion works as follows: $(commands) expands to the output of “commands”. This permits nesting, so commands can include brace expansions.

Backtick expansion expands `commands` to the output of commands (that ` is the key in the upper-left corner of your keyboard).

Here's an example:

#!/bin/bash

files="$(ls)" # a list of the files in the current directory

web\_files=`ls public\_html` # list of files in public\_html directory

echo "$files"

echo "$web\_files"

X=`expr 3 \\* 2 + 4` # expr evaluates arithmetic expressions; “man expr” for details.

echo "$X"

The advantage of the $() substitution method is that it is very easy to nest. However, the backtick substitution is considered slightly more readable, and is supported by even the most basic shells (any #!/bin/sh version is just fine).

**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 .dat file to be analyzed, and the second is what you want the output file to be named. Careful, “analyze.py” only 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 is 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 it has a particular “field” value. Write a script that takes the field value as a command line argument, 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”.**