Bash Scripting Is Fun

To complete a specific task in Linux, you will often find yourself running the same set of commands over and over again. This process can waste a lot of your precious time. In this chapter, you will learn how to create bash scripts so that you can be much more efficient in Linux.

Creating simple scripts

Our first bash script will be a simple script that will output the line "Hello Friend!" to the screen. In Elliot's home directory, create a file named [hello.sh] and insert the following two lines:

```
elliot@ubuntu-linux:~$ cat hello.sh
#!/bin/bash
echo "Hello Friend!"
```

Now we need to make the script executable:

```
elliot@ubuntu-linux:~$ chmod a+x hello.sh
```

And finally, run the script:

```
elliot@ubuntu-linux:~$ ./hello.sh
Hello Friend!
```

Congratulations! You have now created your first bash script! Let's take a minute here and discuss a few things; every bash script must do the following:

- [#!/bin/bash]
- Be executable

You have to insert [#!/bin/bash] at the first line of any bash script; the character sequence [#!] is referred to as a shebang or hashbang and is followed by the path of the bash shell.

The PATH variable

You may have noticed that I used [./hello.sh] to run the script; you will get an error if you omit the leading [./]:

```
elliot@ubuntu-linux:~$ hello.sh
hello.sh: command not found
```

The shell can't find the command [hello.sh]. When you run a command on your terminal, the shell looks for that command in a set of directories that are stored in the [PATH] variable.

You can use the [echo] command to view the contents of your [PATH] variable:

```
elliot@ubuntu-linux:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

The colon character separates the path of each of the directories. You don't need to include the full path of any command or script (or any executable) that resides in these directories. All the commands you have learned so far reside in [/bin] and [/sbin], which are both stored in your [PATH] variable. As a result, you can run the [pwd] command:

```
elliot@ubuntu-linux:~$ pwd
/home/elliot
```

There is no need to include its full path:

```
elliot@ubuntu-linux:~$ /bin/pwd
/home/elliot
```

The good news is that you can easily add a directory to your [PATH] variable. For example, to add [/home/elliot] to your [PATH] variable, you can use the [export] command as follows:

```
elliot@ubuntu-linux:~$ export PATH=$PATH:/home/elliot
```

Now you don't need the leading [./] to run the [hello.sh] script:

```
elliot@ubuntu-linux:~$ hello.sh
Hello Friend!
```

It will run because the shell is now looking for executable files in the [/home/elliot] directory as well:

```
elliot@ubuntu-linux:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/home/elliot
```

Alright! Now let's create a few more bash scripts. We will create a script named [hello2.sh] that prints out "Hello Friend!" then displays your current working directory:

```
elliot@ubuntu-linux:~$ cat hello2.sh
#!/bin/bash
echo "Hello Friend!"
pwd
```

Now let's run it:

```
elliot@ubuntu-linux:~$ hello2.sh
-bash: /home/elliot/hello2.sh: Permission denied
```

Shoot! I forgot to make it executable:

```
elliot@ubuntu-linux:~$ chmod a+x hello2.sh
elliot@ubuntu-linux:~$ ./hello2.sh
Hello Friend!
/home/elliot
```

Reading user input

Let's create a better version of our [hello.sh] script. We will let the user input his/her name and then we will greet the user; create a script named [greet.sh] with the following lines:

```
elliot@ubuntu-linux:~$ cat greet.sh
#!/bin/bash
echo "Please enter your name:"
```

```
read name
echo "Hello $name!"
```

Now make the script executable and then run it:

```
elliot@ubuntu-linux:~$ chmod a+x greet.sh
elliot@ubuntu-linux:~$ ./greet.sh
Please enter your name:
```

When you run the script, it will prompt you to enter your name; I entered [Elliot] as my name:

```
elliot@ubuntu-linux:~$ ./greet.sh
Please enter your name:
Elliot
Hello Elliot!
```

The script greeted me with "Hello Elliot!". We used the [read] command to get the user input, and notice in the [echo] statement, we used a dollar sign, [$], to print the value of the variable [name].

Let's create another script that reads a filename from the user and then outputs the size of the file in bytes; we will name our script [size.sh]:

```
elliot@ubuntu-linux:~$ cat size.sh
#!/bin/bash
echo "Please enter a file path:"
read file
filesize=$(du -bs $file| cut -f1)
echo "The file size is $filesize bytes"
```

And never forget to make the script executable:

```
elliot@ubuntu-linux:~$ chmod a+x size.sh
```

Now let's run the script:

```
elliot@ubuntu-linux:~$ size.sh
Please enter a file path
/home/elliot/size.sh
The file size is 128 bytes
```

I used [size.sh] as the file path, and the output was 128 bytes; is that true? Let's check:

```
elliot@ubuntu-linux:~$ du -bs size.sh
128 size.sh
```

Indeed it is; notice in the script the following line:

```
filesize=$(du -bs $file| cut -f1)
```

It stores the result of the command [du -bs $file | cut -f1] in the variable [filesize]:

```
elliot@ubuntu-linux:~$ du -bs size.sh | cut -f1
128
```

Also notice that the command [du -bs $file cut -f1] is surrounded by parentheses and a dollar sign (on the left); this is called command substitution. In general, the syntax of command substitution goes as follows:

```
var=$(command)
```

The result of the [command] will be stored in the variable [var].

Passing arguments to scripts

Instead of reading input from users, you can also pass arguments to a bash script. For example, let's create a bash script named [size2.sh] that does the same thing as the script [size.sh], but instead of reading the file from the user, we will pass it to the script [size2.sh] as an argument:

```
elliot@ubuntu-linux:~$ cat size2.sh
#!/bin/bash
filesize=$(du -bs $1| cut -f1)
echo "The file size is $filesize bytes"
```

Now let's make the script executable:

```
elliot@ubuntu-linux:~$ chmod a+x size2.sh
```

Finally, you can run the script:

```
elliot@ubuntu-linux:~$ size2.sh /home/elliot/size.sh
The file size is 128 bytes
```

You will get the same output as [size.sh]. Notice that we provided the file path [/home/elliot/size.sh] as an argument to the script [size2.sh].

We only used one argument in the script [size2.sh], and it is referenced by [$1]. You can pass multiple arguments as well; let's create another script [size3.sh] that takes two files (two arguments) and outputs the size of each file:

```
elliot@ubuntu-linux:~$ cat size3.sh #!/bin/bash
filesize1=$(du -bs $1| cut -f1)
filesize2=$(du -bs $2| cut -f1)
echo "$1 is $filesize1 bytes"
echo "$2 is $filesize2 bytes"
```

Now make the script executable and run it:

```
elliot@ubuntu-linux:~$ size3.sh /home/elliot/size.sh /home/elliot/size3.sh
/home/elliot/size.sh is 128 bytes
/home/elliot/size3.sh is 136 bytes
```

Awesome! As you can see, the first argument is referenced by [$1], and the second argument is referenced by [$2]. So in general:

```
bash_script.sh argument1 argument2 argument3 ...
                    $1         $2         $3
```

Using the if condition

You can add intelligence to your bash script by making it behave differently in different scenarios. To do that, we use the conditional [if] statement.

In general, the syntax of the [if condition] is as follows:

```
if [ condition is true ]; then
    do this ...
fi
```

For example, let's create a script [empty.sh] that will examine whether a file is empty or not:

```
elliot@ubuntu-linux:~$ cat empty.sh
#!/bin/bash
filesize=$(du -bs $1 | cut -f1)
if [ $filesize -eq 0 ]; then
echo "$1 is empty!"
fi
```

Now let's make the script executable and also create an empty file named [zero.txt]:

```
elliot@ubuntu-linux:~$ chmod a+x empty.sh
elliot@ubuntu-linux:~$ touch zero.txt
```

Now let's run the script on the file [zero.txt]:

```
elliot@ubuntu-linux:~$ ./empty.sh zero.txt
zero.txt is empty!
```

As you can see, the script correctly detects that [zero.txt] is an empty file; that's because the test condition is true in this case as the file [zero.txt] is indeed zero bytes in size:

```
if [ $filesize -eq 0 ];
```

We used [-eq] to test for equality. Now if you run the script on a non-empty file, there will be no output:

```
elliot@ubuntu-linux:~$ ./empty.sh size.sh
elliot@ubuntu-linux:~$
```

We need to modify the script [empty.sh] so that it displays an output whenever it's passed a non-empty file; for that, we will use the [if-else] statement:

```
if [ condition is true ]; then
    do this ...
else
    do this instead ...
fi
```

Let's edit the [empty.sh] script by adding the following [else] statement:

```
elliot@ubuntu-linux:~$ cat empty.sh
#!/bin/bash
```

```
filesize=$(du -bs $1 | cut -f1)
if [ $filesize -eq 0 ]; then
echo "$1 is empty!"
else
echo "$1 is not empty!"
fi
```

Now let's rerun the script:

```
elliot@ubuntu-linux:~$ ./empty.sh size.sh
size.sh is not empty!
elliot@ubuntu-linux:~$ ./empty.sh zero.txt
zero.txt is empty!
```

As you can see, it now works perfectly!

You can also use the [elif] (**else-if**) statement to create multiple test conditions:

```
if [ condition is true ]; then
    do this ...
elif [ condition is true]; then
    do this instead ...
fi
```

Let's create a script [filetype.sh] that detects a file type. The script will output whether a file is a regular file, a soft link, or a directory:

```
elliot@ubuntu-linux:~$ cat filetype.sh
#!/bin/bash
file=$1
if [ -f $1 ]; then
echo "$1 is a regular file"
elif [ -L $1 ]; then
echo "$1 is a soft link"
elif [ -d $1 ]; then
echo "$1 is a directory"
fi
```

Now let's make the script executable and also create a soft link to [/tmp] named [tempfiles]:

```
elliot@ubuntu-linux:~$ chmod a+x filetype.sh
elliot@ubuntu-linux:~$ ln -s /tmp tempfiles
```

Now run the script on any directory:

```
elliot@ubuntu-linux:~$ ./filetype.sh /bin
/bin is a directory
```

It correctly detects that [/bin] is a directory. Now run the script on any regular file:

```
elliot@ubuntu-linux:~$ ./filetype.sh zero.txt
zero.txt is a regular file
```

It correctly detects that [zero.txt] is a regular file. Finally, run the script on any soft link:

```
elliot@ubuntu-linux:~$ ./filetype.sh tempfiles
tempfiles is a soft link
```

It correctly detects that [tempfiles] is a soft link.

The following [man] page contains all the test conditions:

```
elliot@ubuntu-linux:~$ man test
```

So NEVER memorize! Utilize and make use of the man pages.

Looping in bash scripts

The ability to loop is a very powerful feature of bash scripting. For example, let's say you want to print out the line "Hello world" 20 times on your terminal; a naive approach would be to create a script that has 20 [echo] statements. Luckily, looping offers a smarter solution.

Using the for loop

The [for] loop has a few different syntaxes. If you are familiar with C++ or C programming, then you will recognize the following [for] loop syntax:

```
for ((initialize ; condition ; increment)); do
// do something
done
```

Using the aforementioned C-style syntax; the following [for] loop will print out "Hello World" twenty times:

```
for ((i = 0 ; i < 20 ; i++)); do
    echo "Hello World"
done
```

The loop initializes the integer variable [i] to [0], then it tests the condition ([i < 20]); if true, it then executes the line echo "Hello World" and increments the variable [i] by one, and then the loop runs again and again until [i] is no longer less than [20].
Now let's create a script [hello20.sh] that has the [for] loop we just discussed:

```
elliot@ubuntu-linux:~$ cat hello20.sh
#!/bin/bash
for ((i = 0 ; i < 20 ; i++)); do
    echo "Hello World"
done
```

Now make the script executable and run it:

```
elliot@ubuntu-linux:~$ chmod a+x hello20.sh
elliot@ubuntu-linux:~$ hello20.sh
Hello World
Hello World
Hello World
Hello World
Hello World
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

It outputs the line "Hello World" twenty times as we expected. Instead of the C-style syntax, you can also use the range syntax with the [for] loop:

```
for i in {1..20}; do
    echo "Hello World"
done
```

This will also output "Hello World" 20 times. This range syntax is particularly useful when working with a list of files. To demonstrate, create the following five files:

```
elliot@ubuntu-linux:~$ touch one.doc two.doc three.doc four.doc five.doc
```

Now let's say we want to rename the extension for all five files from [.doc] to [.document]. We can create a script [rename.sh] that has the following [for] loop:

```
#!/bin/bash
for i in /home/elliot/*.doc; do
    mv $i $(echo $i | cut -d. -f1).document
done
```

Make the script executable and run it:

```
#!/bin/bash
elliot@ubuntu-linux:~$ chmod a+x rename.sh
elliot@ubuntu-linux:~$ ./rename.sh
elliot@ubuntu-linux:~$ ls *.document
five.document four.document one.document three.document two.document
```

As you can see, it renamed all the files with the [.doc] extension to [.document]. Now imagine if you wanted to do this for a million files. If you don't know bash scripting, you would probably spend ten years doing it. We should all thank the Linux Gods for bash scripting.

Using the while loop

The [while] loop is another popular and intuitive loop. The general syntax for a [while] loop is as follows:

```
while [ condition is true ]; do
  // do something
done
```

For example, we can create a simple script [numbers.sh] that prints the numbers from one to ten:

```
elliot@ubuntu-linux:~$ cat numbers.sh
#!/bin/bash
number=1
while [ $number -le 10 ]; do
echo $number
number=$(($number+1))
done
```

Make the script executable and run it:

```
elliot@ubuntu-linux:~$ chmod a+x numbers.sh
elliot@ubuntu-linux:~$ ./numbers.sh
1
2
3
4
5
6
7
8
9
10
```

The script is simple to understand; we first initialized the variable number to [1]:

```
number=1
```

Then we created a test condition that will keep the [while] loop running as long as the variable [number] is less than or equal to 10:

```
while [ $number -le 10 ]; do
```

Inside the body of the [while] loop, we first print out the value of the variable [number], and then we increment it by one. Notice that to evaluate an arithmetic expression, it needs to be within double parentheses as [$((arithmetic-expression))]:

```
echo $number
number=$(($number+1))
```

Now it's time for some fun! We will create a number guessing game. But before we do that, let me introduce you to a pretty cool command. You can use the shuffle command [shuf] to generate random permutations. For example, to generate a random permutation of the numbers between 1 and 10, you can run the following command:

```
elliot@ubuntu-linux:~$ shuf -i 1-10
1
6
5
```

```
2
10
8
3
9
7
4
```

Keep in mind that my output will most likely be different from your output because it is random! There is a one in a million chance that you will have the same output as me.

Now we can use the [-n] option to select one number out of the permutation. This number will be random as well. So to generate a random number between 1 and 10, you can run the following command:

```
elliot@ubuntu-linux:~$ shuf -i 1-10 -n 1
6
```

The output will be a random number between 1 and 10. The [shuf] command will play a key role in our game. We will generate a random number between 1 and 10, and then we will see how many tries it will take the user (player) to guess the random number correctly.

Here is our lovely handcrafted script [game.sh]:

```
elliot@ubuntu-linux:~$ cat game.sh
#!/bin/bash
random=$(shuf -i 1-10 -n 1) #generate a random number between 1 and 10.
echo "Welcome to the Number Guessing Game"
echo "The lucky number is between 1 and 10."
echo "Can you guess it?"
tries=1
while [ true ]; do
echo -n "Enter a Number between 1-10: "
read number
if [ $number -gt $random ]; then
echo "Too high!"
elif [ $number -lt $random ]; then
echo "Too low!"
else
echo "Correct! You got it in $tries tries"
break #exit the loop
fi
tries=$(($tries+1))
done
```

Now make the script executable and run it to start the game:

```
elliot@ubuntu-linux:~$ chmod a+x game.sh
elliot@ubuntu-linux:~$ game.sh
Welcome to the Number Guessing Game
The lucky number is between 1 and 10.
Can you guess it?
Enter a Number between 1-10: 4
Too low!
Enter a Number between 1-10: 7
```

```
Too low!
Enter a Number between 1-10: 9
Too high!
Enter a Number between 1-10: 8
Correct! You got it in 4 tries
```

It took me four tries in my first attempt at the game; I bet you can easily beat me!
Let's go over our game script line by line. We first generate a random number between 1 and 10 and assign it to the variable [random]:

```
random=$(shuf -i 1-10 -n 1) #generate a random number between 1 and 10.
```

Notice that you can add comments in your bash script as I did here by using the hash character, followed by your comment.

We then print three lines that explain the game to the player:

```
echo "Welcome to the Number Guessing Game"
echo "The lucky number is between 1 and 10."
echo "Can you guess it?"
```

Next, we initialize the variable [tries] to [1] so that we can keep track of how many guesses the player took:

```
tries=1
```

We then enter the game loop:

```
while [ true ]; do
```

Notice the test condition [while [true]] will always be [true], and so the loop will keep running forever (infinite loop).

The first thing we do in the game loop is that we ask the player to enter a number between 1 and 10:

```
echo -n "Enter a Number between 1-10: "
read number
```

We then test to see if the number the player has entered is greater than, less than, or equal to the [random] number:

```
if [ $number -gt $random ]; then
echo "Too high!"
elif [ $number -lt $random ]; then
echo "Too low!"
else
echo "Correct! You got it in $tries tries"
break #exit the loop
fi
```

If [number] is bigger than [random], we tell the player that the guess is too high to make it easier for the player to have a better guess next time. Likewise, if [number] is smaller than [random], we tell the player the guess is too low. Otherwise, if it is a correct guess, then we print the total number of tries the player exhausted to make the correct guess, and we break from the loop.

Notice that you need the [break] statement to exit from the infinite loop. Without the [break] statement, the loop will run forever.

Finally, we increment the number of [tries] by 1 for each incorrect guess (high or low):

```
tries=$(($tries+1))
```

I have to warn you that this game is addictive! Especially when you play it with a friend to see who will get the correct guess in the least number of tries.

Using the until loop

Both the [for] and [while] loops run as long as the test condition is [true]. On the flip side, the [until] loop keeps running as long as the test condition is [false]. That's to say, it stops running as soon as the test condition is [true].

The general syntax of an [until] loop is as follows:

```
until [condition is true]; do
  [commands]
done
```

For example, we can create a simple script [3x10.sh] that prints out the first ten multiples of [3]:

```
elliot@ubuntu-linux:~$ cat 3x10.sh
#!/bin/bash
counter=1
until [ $counter -gt 10 ]; do
echo $(($counter * 3))
counter=$(($counter+1))
done
```

Now make the script executable and then run it:

```
elliot@ubuntu-linux:~$ chmod a+x 3x10.sh
elliot@ubuntu-linux:~$ 3x10.sh
3
6
9
12
15
18
21
24
27
30
```

The script is easy to understand, but you might scratch your head a little bit trying to understand the test condition of the [until] loop:

```
until [ $counter -gt 10 ]; do
```

The test condition basically says: "until [counter] is greater than 10, keep running!"

Notice that we can achieve the same result with a [while] loop that has the opposite test condition. You simply negate the test condition of the [until] loop and you will get the [while] loop equivalent:

```
while [ $counter -le 10 ]; do
```

In mathematics, the opposite (negation) of greater than ([>]) is less than or equal to ([≤]). A lot of people forget the [equal to] part. Don't be one of those people!

Bash script functions

When your scripts get bigger and bigger, things can get very messy. To overcome this problem, you can use bash functions. The idea behind functions is that you can reuse parts of your scripts, which in turn produces better organized and readable scripts.

The general syntax of a bash function is as follows:

```
function_name () {
<commands>
}
```

Let's create a function named [hello] that prints out the line "Hello World". We will put the [hello] function in a new script named [fun1.sh]:

```
elliot@ubuntu-linux:~$ cat fun1.sh
#!/bin/bash

hello () {
echo "Hello World"
}

hello     # Call the function hello()
hello     # Call the function hello()
hello     # Call the function hello()
```

Now make the script executable and run it:

```
elliot@ubuntu-linux:~$ chmod a+x fun1.sh
elliot@ubuntu-linux:~$ ./fun1.sh
Hello World
Hello World
Hello World
```

The script outputs the line "Hello World" three times to the terminal. Notice that we called (used) the function [hello] three times.

Passing function arguments

Functions can also take arguments the same way a script can take arguments. To demonstrate, we will create a script [math.sh] that has two functions [add] and [sub]:

```
elliot@ubuntu-linux:~$ cat math.sh
#!/bin/bash

add () {
echo "$1 + $2 =" $(($1+$2))
}
```

```
sub () {
echo "$1 - $2 =" $(($1-$2))
}

add 7 2
sub 7 2
```

Make the script executable and then run it:

```
elliot@ubuntu-linux:~$ chmod a+x math.sh
elliot@ubuntu-linux:~$ ./math.sh
7 + 2 = 9
7 - 2 = 5
```

The script has two functions [add] and [sub]. The [add] function calculates and outputs the total of any given two numbers. On the other hand, the [sub] function calculates and outputs the difference of any given two numbers.

No browsing for you

We will conclude this chapter with a pretty cool bash script [noweb.sh] that makes sure no user is having fun browsing the web on the Firefox browser:

```
elliot@ubuntu-linux:~$ cat noweb.sh
#!/bin/bash

shutdown_firefox() {
killall firefox 2> /dev/null
}

while [ true ]; do
shutdown_firefox
sleep 10 #wait for 10 seconds
done
```

Now open Firefox as a background process:

```
elliot@ubuntu-linux:~$ firefox &
[1] 30436
```

Finally, make the script executable and run the script in the background:

```
elliot@ubuntu-linux:~$ chmod a+x noweb.sh
elliot@ubuntu-linux:~$ ./noweb.sh &
[1] 30759
```

The moment you run your script, Firefox will shut down. Moreover, if you run the script as the [root] user, none of the system users will be able to enjoy Firefox!

Knowledge check

For the following exercises, open up your terminal and try to solve the following tasks:

1. Create a bash script that will display the calendar of the current month.
2. Modify your script so it displays the calendar for any year (passed as an argument).
3. Modify your script so it displays the calendar for all the years from [2000] to [2020].