Lecture 4: Writing shell scripts

Your first shell script

Shell scripts are nothing other than files that contain shell commands that are run when you type the file at the command line. That is, the commands that make up a shell script are identical to those you might type at the command prompt, except that they are conveniently stored in a file so that you can execute them over and over again without having to type the commands every time.

In order to start editing a shell script, you need to open up a file for editing. You can do so with the `emacs` editor. Let's say we want to write a very simple shell script called `simple.sh` that prints the following message to the screen:

```
This shell script was written by me.
```

To do so, open up the `emacs` edtior with

```
$ emacs -nw simple.sh
```

The emacs editor will appear, and you should enter the following text

```
#
# My first shell script
#
echo This shell script was written by me.
```

Now, save your file with `C-x C-s`, and suspend emacs with `C-z`. You now have a shell script called `simple.sh` in your current directory. The `#` are comments, and anything after a `#` is ignored by the shell. There are two ways you can execute this script. The first is by "sourcing" the script, which just means you want to run the script as if you are typing the commands within it in your current shell. To do so, you just type

```
$ . ./simple.sh
This shell script was written by me.
```

All this did was send the lines in your file except for those that started with the `#` to the current shell, which executed the `echo` command and printed the result to the screen. So running the script is identical to typing the following at the command prompt

```
$ #
$ # My first shell script
$ #
$ echo This shell script was written by me.
This shell script was written by me.
```

From this you can see that typing `#` at the prompt and hitting the return key does nothing.

The other way you can execute your script is to make it executable and use it as a command. To make this script executable, use the `chmod` command

```
$ chmod u+x simple.sh
```

Once the file is executable by the user, you can execute it by typing it at the command line with

```
$ ./simple.sh
This shell script was written by me.
```

Note that this will only work if the permissions on the file are executable by the user. We have to type `./simple.sh` because the local directory “`.`“ is not in our search path. This method of running your shell script is identical to typing

```
$ bash ./simple.sh
This shell script was written by me.
```

which tells the shell to run the program `bash` and have it read the contents of the file `simple.sh`. The program `bash` is the program that is reading the commands as you type them in right now.

Sometimes your shell script might be run by someone who prefers to use a different shell, like `tcsh`. If they tried to run your script at the command line, it might fail because some of the syntax is different between the two languages. When using a different shell, if the user specifies the correct shell at the command line with

```
$ bash ./simple.sh
```

then it doesn't matter which shell is running the command prompt. However, a better way to do it is to specify which shell to run within your script. The way to do this is to include the line `#!/bin/bash` at the beginning of your script, so that now your script looks like

```
#!/bin/bash
#
# My first shell script
#
echo This shell script was written by me.
```

Now when you type `./simple.sh` at the command line, the results will look the same, but the first line guarantees that a `bash` shell will open up and interpret the script, even if the calling shell is not `bash`. The first line of this script is an exception to the comment rule. If this line were placed at any other line in your script, it would be viewed as a comment.

Running your scripts with command line arguments

Now let's say we want to add a bit more functionality to our script. Let's say we would like to run our command but have it print out the name we supply to it. For example, let's say we wanted the command to work as follows

```
$ ./simple.sh Oliver
This shell script was written by Oliver.
```

To have the script read the command line arguments, you would use `$1` to refer to the first argument provided at the command line, so that your script now looks like

```
#!/bin/bash
#
# My first shell script
#
echo This shell script was written by $1.
```

Now your script will print out

```
This shell script was written by Oliver.
```

when it is called with `./simple.sh Oliver`. You can add several arguments and treat them all accordingly. For example, if your script looked like this

```
#!/bin/bash
#
# My first shell script
#
echo This shell script was written by $1.
echo I have written $2 scripts so far.
```

and you ran it with two arguments, you would get

```
$ ./simple.sh Oliver 2
This shell script was written by Oliver.
I have written 2 scripts so far.
```

If you ran this script with no arguments, you would get

```
$ ./simple.sh
This shell script was written by .
I have written scripts so far.
```

In the next section, we will deal with learning how to use `if` statements to check whether or not there are enough arguments and whether or not the supplied arguments make sense.

If statements

Comparing integers

In the script as it is so far, we can run it with as many arguments as we want. We already showed that it could be run with no arguments, but if we ran it with 5 arguments it would also work, but the last three would not be used, such as

```
$ ./simple.sh Oliver 2 COS315 UNIX 5
This shell script was written by Oliver.
I have written 2 scripts so far.
```

We can make sure that there are enough arguments supplied to the script by using the `if` statement. The number of arguments supplied to the script is given by the `$#` character, which we can use in the script by adding the line

```
#!/bin/bash
#
# My first shell script
#
echo This script has been called with $# arguments.

echo This shell script was written by $1.
echo I have written $2 scripts so far.
```

If we run this script with the five arguments, we will get

```
$ ./simple.sh Oliver 2 COS315 UNIX 5
This script has been called with 5 arguments.
This shell script was written by Oliver.
I have written 2 scripts so far.
```

Clearly we do not want the script to run when more than two arguments have been called. We can check to see if `$#` is greater than 2 with the `if` statement, and if it is, then we should exit the script with a statement that tells us the correct way to use it. The `if` statement that would do this for us is given in the script as

```
#!/bin/bash
#
# My first shell script
#
if [ $# -gt 2 ] ; then
  echo Usage: ./simple.sh name times
  exit 1;
fi
echo This script has been called with $# arguments.

echo This shell script was written by $1.
echo I have written $2 scripts so far.
```

so that when we run the script we will get

```
$ ./simple.sh Oliver 2 COS315 UNIX 5
Usage: ./simple.sh name times
```

Now let's say we want to also check to see that there are at least two arguments to the script. This is done by editing the `if` statement so that it reads as

```
if [ $# -gt 2 ] ; then
  echo Usage: ./simple.sh name times
  exit 1;
elif [ $# -le 0 ] ; then
  echo Usage: ./simple.sh name times
  exit 1;
fi
```

so now if we try and use the script with no arguments then we will get

```
$ ./simple.sh
Usage: ./simple.sh name times
```

When you are comparing two integers `n1` and `n2` using if statements, as we are doing, you use the following

`n1 -eq n2`	true when `n1` is equal to `n2`
`n1 -ne n2`	true when `n1` is not equal to `n2`
`n1 -lt n2`	true when `n1` is less than `n2`
`n1 -gt n2`	true when `n1` is greater than `n2`
`n1 -le n2`	true when `n1` is less than or equal to `n2`
`n1 -ge n2`	true when `n1` is greater than or equal to `n2`

In the script we used the `exit` command to break out of the program when an error was encountered. The particular exit code, or number, that you supply with the `exit` command is completely arbitrary, except it is standard to have a code exit with an exit code of `0` when no errors have been encountered. The point of the exit code is so that you can determine why your script failed by looking at what the exit code was. For example, given the simple script

```
#!/bin/bash

exit 5;
```

If we call this script `error.sh` and run it, nothing is printed to the screen. However, we can determine what its exit code was by looking at the `$?` variable:

```
$ ./error.sh
$ echo $?
5
```

If you look at the `$?` variable again, it will be `0`, since the `echo` command exited without errors and the `#?` variable shows the exit code from the last command that was run. Exit codes are useful for debugging very complicated scripts which list the error codes in the documentation for the script. You should employ error codes when you write the script for programming assignment 1.

Compound if statements

Rather than using two if statements to test both conditions on the arguments, we can use compound if statements as in

```
if [ $# -gt 2 ] || [ $# -le 0 ] ; then
  echo Usage: ./simple.sh name times
fi
```

which is identical to using two `if` statements except that it is more compact. Here, the `||` characters imply the "or" logical operator. The "or" logical operator can also be used in the form

```
if [ $# -gt 2 -o $# -le 0 ] ; then
  echo Usage: ./simple.sh name times
fi
```

We can also use the "and" logical operator with is given by either

```
[ $# -gt 2 -a $# -le 0 ]
```

or

```
[ $# -gt 2 ] && [ $# -le 0 ]
```

You should note that both of these statements would never be true since `$#` could never be greater than 2 and less than or equal to 0 at the same time, but these are used only to illustrate the syntax.

Checking existence of files or directories

Now let's say we want to use the `if` statement to test whether or not a directory we supply at the command line exists. This is done using

```
if [ -d $1 ] ; then
  echo The directory exists!
else
  echo The directory does not exist!
fi
```

If these lines are used in a shell script, which is called `test.sh` and run, then we get

```
$ ./test.sh /home/ofringer
The directory exists!
```

We can also use this syntax to test the existence of regular files (not block or character files, which are tested with `-b` and `-c`) with the `-f` option instead of `-d`. If we want do check and see if a given argument is not a file, we use

```
if [ ! -f $1 ] ; then
  echo The file does not exist!
else
  echo The file does exist!
fi
```

Comparing strings

The `if` statement can also be used to compare strings. For example, let's say we wanted to check and see if a command line argument matched the string "info". This would be done with

```
str=''info''
if [ $1 = $str ] ; then
  echo The string matches.
fi
```

We can also check to see if it does not match it with `!=`. In order to check and see if strings are empty, we use the `-z` option, as in

```
str=''info''
if [ -z $1 ] ; then
  echo The string is empty!
fi
```

For loops

For loops are used to execute commands on each file in a given list. For example, if we wanted our `test.sh` script to list the jpeg files in the directory provided , say the images directory from `/home/cos315/assignments/assignment1/images`, our shell script would look something like

```
#!/bin/bash
for file in $1/*.jpg
do
  echo $file
done
```

So that when we run the script we get

```
$ ./test.sh images
images/img_5974.jpg
images/img_6005.jpg
images/img_6017.jpg
images/img_6288.jpg
images/img_6345.jpg
images/img_6366.jpg
images/img_6764.jpg
images/img_6830.jpg
images/img_7435.jpg
images/img_7461.jpg
images/img_7658.jpg
images/img_7731.jpg
```

Now let's say we wanted to count the number of files in the directory. In order to do so, we need to use the `expr` command, which evaluates arithmetic expressions. For example, at the command line, if we wanted to add two numbers `x` and `y`, we would type

```
$ x=3;y=2;expr $x + $y
5
```

We can also subtract two numbers with

```
$ x=3;y=2;expr $x - $y
5
```

The `expr` command can then be used in the for loop to count the number of files in the directory with

```
#!/bin/bash
n=0
for file in $1/*.jpg
do
  n=‘expr $n + 1‘
done
echo There are $n files in the directory $1.
```

so that typing the script at the command line would result in

```
$ ./test.sh images
There are 12 files in the directory images.
```

Of course, we can use other already existing functions to count the number of files in the directory `images` with

```
$ ls images/*.jpg | wc -w
     12
```

Which we can in turn use to test if there are any jpeg files in the directory with

```
numfiles=`ls $1/*.jpg | wc -w`
if [ $numfiles -eq 0 ] ; then
  echo There are no jpeg files in the directory $1!
fi
```

So, using the above as a script, if we made an empty directory with `mkdir empty`, and tried to count the number of jpg files in that directory, we would get

```
$ ./test.sh empty
ls: empty/*.jpg: No such file or directory
There are no jpeg files in the directory empty.
```

In order to remove the error with the `ls` command, we could redirect its standard error into the UNIX trash file `/dev/null` with

```
numfiles=`ls $1/*.jpg 2> /dev/null | wc -w`
if [ $numfiles -eq 0 ] ; then
  echo There are no jpeg files in the directory $1!
fi
```

so that we do not see the error associated with `ls` when it doesn't find the specified file. The `/dev/null` file is a special file that you can use to redirect messages to that you do not want to appear on the screen.

We can employ `if` statements within the `for` loop if we would like to list out the files in a specified number of groups with a specific number in each group. If the number of groups was the second argument provided at the command line, then the script to do so would be given by

```
#!/bin/bash

dir=$1
gps=$2

n=1
for file in $dir/*.jpg
do
  echo $file
  if [ $n -eq $gps ] ; then
    n=1
    echo ;
  else
    n=`expr $n + 1`
  fi
done
```

So that calling this script would yield

```
$ ./test.sh images 4
images/img_5974.jpg
images/img_6005.jpg
images/img_6017.jpg
images/img_6288.jpg

images/img_6345.jpg
images/img_6366.jpg
images/img_6764.jpg
images/img_6830.jpg

images/img_7435.jpg
images/img_7461.jpg
images/img_7658.jpg
images/img_7731.jpg
```

Which shows that the script prints out the images in the `images` directory in three groups, each group listing four files.