Introducing Unix Shell Scripting Script

Devasena Inupakutika, University of Southampton, The Software Sustainability Institute, April 2014

Things you should do are written in bold.

Suggested dialog is in normal text.

Command-line excerpts and code fragments are in shaded fixed-width font.

Introduction

What is the shell and how do I access it?

The *shell* is a program that presents a command line interface, which allows you to control your computer using commands entered with a keyboard instead of controlling graphical user interfaces (GUIs) with a mouse/keyboard combination.

Use a browser to open the tutorial on github, located at: http://github.com/{{page.github_username}}/{{page.bootcamp_slug}}

Click on the directory named /lessons/nocs/nocs-shell.

A *terminal* is a program you run that gives you access to the shell. There are many different terminal programs that vary across operating systems.

There are many reasons to learn about the shell. In my opinion, the most important reasons are that:

1. It is very common to encounter the shell and command-line-interfaces in scientific computing, so you will probably have to learn it eventually

2. The shell is a really powerful way of interacting with your computer. GUIs and the shell are complementary - by knowing both you will greatly expand the range of tasks you can accomplish with your computer. You will also be able to perform many tasks more efficiently.

    The shell is just a program and there are many different shell programs that have been developed. The most common shell (and the one we will use) is called the Bourne-Again SHell (bash). Even if bash is not the default shell, it is usually installed on most systems and can be started by typing `bash` in the terminal. Many commands, especially a lot of the basic ones, work across the various shells but many things are different. I recommend sticking with bash and learning it well. ([Here is a link for more information](#)). Shell script allows us to program commands in chains and have the system execute them as a scripted event similar to batch files.

To open a terminal, just single click on the "Terminal" icon on the Desktop.

The Example: Manipulating Experimental Data Files

We will spend most of our time learning about the basics of the shell by manipulating some experimental altimetry data from Global Ocean tide solution models. To get data for this test, you'll need Internet access. Just enter the command:

```
git clone https://github.com/{{page.github_username}}/{{page.bootcamp_slug}}.git
```

Followed by:

```
cd {{page.bootcamp_slug}}
```

These 2 commands will grab all of the data needed for this workshop from the Internet.

Let's get started

One very basic command is `echo`. This command just prints text to the terminal. Try the command:

```
echo Hello, World
```

Then press enter. You should see the text "Hello, World" printed back to you. The echo command is useful for printing from a shell script, for displaying variables, and for generating known values to pass to other programs.

Files and Directories

Moving around the file system

Let's learn how to move around file system using command line scripts. This is really easy to do in GUI. But, once you learn basic Unix commands, you'll see that it is pretty easy to do in shell too.

For this purpose, we need to know our current location. The command `pwd` (print working directory) tells you where you are sitting in the directory tree. The command `ls` will list the files in files in the current directory. Directories are often called "folders" because of how they are represented in GUIs. Directories are just listings of files. They can contain other files or directories.

Several commands are frequently used to create, inspect, rename and delete files and directories. In order to explore them, let's open shell window, which starts usually with a dollar sign prompt indicating that shell is waiting for input:

```
$
```

Whenever you start up a terminal, you will start in a special directory called the *home* directory. Every user has their own home directory where they have full access to do whatever they want. For example, if our user ID is `user`, the `pwd` command tells us that we are in the `/home/user` directory. At any moment, our current working directory is our current default directory, i.e., the directory that the computer assumes we want to run commands in unless we explicitly specify something else. Here, the computer's response is `/users/user`, which is user's home directory:

```
$ pwd
/users/user
```

This is the home directory for the `user` user. That is our user name. You can always find out your user name by entering the command `whoami`.

```
$ whoami
user
```

More specifically, when we type `whoami` the shell:

1. Finds a program called `whoami`,
2. Runs that program,
3. Displays that program's output, then
4. Displays a new prompt to tell us that it's ready for more commands.

File types

When you enter the `ls` command lists the contents of the current directory. There are several items in the home directory, notice that they are all coloured blue. This tells us that all of these items are directories as opposed to files.

Lets create an empty file using the `touch` command. Enter the command:

```
touch testfile
```

Then list the contents of the directory again. You should see that a new entry, called `testfile`, exists. It is coloured white meaning that it is a file, as opposed to a directory. The `touch` command just creates an empty file.

Some terminals will not colour the directory entries in this very convenient way. In those terminals, use `ls -F` instead of `ls`. The `-F` argument modifies the results so that a slash is placed at the end of directories. If the file is *executable* meaning that it can be run like a program, then a star will be placed at the end of the file name.

You can also use the command `ls -l` to see whether items in a directory are files or directories. `ls -l` gives a lot more information too, such as the size of the file and information about the owner. If the entry is a directory, then the first letter will be a

"d". The fifth column shows you the size of the entries in bytes. Notice that testfile has a size of zero.

Now, let's get rid of testfile. To remove a file, just enter the command:

```
rm testfile
```

The rm command can be used to remove files. If you enter ls again, you will see that testfile is gone.

Changing directories

Now let's move to a different directory. The command cd (change directory) is used to move around. Let's move into the {{page.bootcamp_slug}} directory. Enter the following command:

```
cd {{page.bootcamp_slug}}
```

Now use the ls command to see what is inside this directory. You will see that there is an entry, which is green. This means that this is an executable. If you use ls -F you will see that this file ends with a star.

This directory contains all of the material for this bootcamp. Now move to the directory containing the data for the shell tutorial which is present at /lessons/nocs/nocs-shell :

```
cd nocs-shell
```

If you enter the cd command by itself, you will return to the home directory. Try this, and then navigate back to the nocs-shell directory.

cd doesn't print anything, but if we run pwd after it, we can see the path of the directory we're in currently. If we run ls without parameters now, it lists the contents of current directory.

We now know how to go down the directory tree: How do we go up? We could use an absolute path:

```
$ cd /users/user
```

but it's almost always simpler to use `cd ..` to go up one level:

```
$ cd ..
```

`..` is a special directory name meaning "the directory containing this one", or more succinctly, the parent of the current directory. Sure enough, if we run `pwd` after running `cd ..`, we're back in `/users/user`:

```
$ pwd
/users/user
```

The special directory `..` doesn't usually show up when we run `ls`. If we want to display it, we can give `ls` the `-a` flag:

```
$ ls -F -a
./      ../   <rest of the files/ directories>
```

`-a` stands for "show all"; it forces `ls` to show us file and directory names that begin with `.`, such as `..` (which, if we're in `/users/user`, refers to the `/users` directory). As you can see, it also displays another special directory that's just called `.`, which means "the current working directory". It may seem redundant to have a name for it, but we'll see some uses for it soon.

Arguments

Most programs take additional arguments that control their exact behaviour. For example, `-F` and `-l` are arguments to `ls`. The `ls` program, like many programs, takes a lot of arguments. But how do we know what the options are to particular commands?

Most commonly used shell programs have a manual. You can access the manual using the man program. Try entering:

```
man ls
```

This will open the manual page for ls. Use the space key to go forward and b to go backwards. When you are done reading, just hit q to exit.

Programs that are run from the shell can get extremely complicated. To see an example, open up the manual page for the find program, which we will use later this session. No one can possibly learn all of these arguments, of course. So you will probably find yourself referring back to the manual page frequently.

Examining the contents of other directories

By default, the ls command lists the contents of the working directory (i.e. the directory you are in). You can always find the directory you are in using the pwd command. However, you can also give ls the names of other directories to view. Navigate to the home directory if you are not already there. Then enter the command:

```
ls {{page.bootcamp_slug}}
```

This will list the contents of the {{page.bootcamp_slug}} directory without you having to navigate there. Now enter:

```
ls {{page.bootcamp_slug}}/lessons/nocs/nocs-shell
```

This prints the contents of nocs-shell. The cd command works in a similar way. Try entering:

```
cd {{page.bootcamp_slug}}/ lessons/nocs/nocs-shell
```

and you will jump directly to nocs-shell without having to go through the intermediate directory.

Full vs. Relative Paths

The `cd` command takes an argument, which is the directory name. Directories can be specified using either a *relative* path a full *path*. The directories on the computer are arranged into a hierarchy. The absolute path tells you where a directory is in that hierarchy. Navigate to the home directory. Now, enter the `pwd` command and you should see:

```
/home/user
```

which is the full name of your home directory. This tells you that you are in a directory called `user`, which sits inside a directory called `home` which sits inside the very top directory in the hierarchy. The very top of the hierarchy is a directory called `/` which is usually referred to as the *root directory*. So, to summarize: `user` is a directory in `home` which is a directory in `/`.

Now enter the following command:

```
cd /home/user/{{page.bootcamp_slug}}/lessons/nocs/nocs-shell
```

This jumps to `nocs-shell`. Now go back to the home directory. We saw earlier that the command:

```
cd {{page.bootcamp_slug}}/lessons/nocs/nocs-shell
```

had the same effect - it took us to the `nocs-shell` directory. But, instead of specifying the absolute path (`/home/user/{{page.bootcamp_slug}}/lessons/nocs/nocs-shell`), we specified a *relative path*. In other words, we specified the path relative to our current directory. An absolute path always starts with a `/`. A relative path does not. You can usually use either an absolute path or a relative path depending on what is most convenient. If we are in the home directory, it is more convenient to just enter the relative path since it involves less typing.

Now, list the contents of the /bin directory. Do you see anything familiar in there?

Saving time with shortcuts, wild cards, and tab completion

Shortcuts

There are some shortcuts, which you should know about. Dealing with the home directory is very common. So, in the shell the tilde character, ~, is a shortcut for your home directory. Navigate to the nocs-shell directory, then enter the command:

```
ls ~
```

This prints the contents of your home directory, without you having to type the absolute path. The shortcut .. always refers to the directory above your current directory. Thus:

```
ls ..
```

prints the contents of the /home/user/{{page.bootcamp_slug}}. You can chain these together, so:

```
ls ../../
```

prints the contents of /home/user which is your home directory. Finally, the special directory . always refers to your current directory. So, ls, ls ., and ls ./././. all do the same thing, they print the contents of the current directory. This may seem like a useless shortcut right now, but we'll see when it is needed in a little while.

To summarize, the commands ls ~, ls ~/., ls ../../, and ls /home/user all do exactly the same thing. These shortcuts are not necessary, they are provided for your convenience.

Our data set: Cochlear Implants and NOCS sample data

A cochlear implant is a small electronic device that is surgically implanted in the inner ear to give deaf people a sense of hearing. More than a quarter of a million people have them, but there is still no widely accepted benchmark to measure their effectiveness. In order to establish a baseline for such a benchmark, our supervisor got teenagers with CIs to listen to audio files on their computer and report:

1. The quietest sound they could hear

2. The lowest and highest tones they could hear
3. The narrowest range of frequencies they could discriminate

To participate, subjects attended our laboratory and one of our lab techs played an audio sample, and recorded their data - when they first heard the sound, or first heard a difference in the sound. Each set of test results was written out to a text file, one set per file. Each participant has a unique subject ID, and a made-up subject name. Each experiment has a unique experiment ID. The experiment has collected 351 files so far.

Another experimental oceanography data folder `nocs` exists which is based on estimated abstractions from tidal waters and all other sources.

The data is a bit of a mess! There are inconsistent file names, there are extraneous "NOTES" files that we'd like to get rid of, and the data is spread across many directories. We are going to use shell commands to get this data into shape. By the end we would like to:

1. Put all of the data into one directory called "alldata"
2. Have all of the data files in there, and ensure that every file has a ".txt" extension
3. Get rid of the extraneous "NOTES" files

If we can get through this example in the available time, we will move onto more advanced shell topics...

Wild cards:

Navigate to the `nocs-shell/data/nocs` directory. This directory contains Estimated abstractions data from tidal waters and all other sources by purpose and environment agency region for a period 2000 – 2010. Try this command:

```
ls *.csv
```

lists files in `nocs-shell/data/nocs` directory that ends with characters .csv.

`nocs-shell/data/thomas` directory contains our hearing test data for Thomas. If we type `ls`, we will see that there are a bunch of files that are just four digit numbers. By default, `ls` lists all of the files in a given directory. The `*` character is a shortcut for "everything". Thus, if you enter `ls *`, you will see all of the contents of a given directory. Now try this command:

```
ls ../thomas/*1
```

This lists every file in `nocs-shell/data/thomas` directory that ends with a `1`. This command:

```
ls /usr/bin/*.sh
```

Lists every file in `/usr/bin` that ends in the characters `.sh`. And this command:

```
ls *2*.csv
```

lists every file in the current directory which contains the number `2`, and ends with characters `.csv`. There are two such files: `Allsource-201205.csv` and `Tidal-EA-201205.csv`.

So how does this actually work? Well...when the shell (bash) sees a word that contains the `*` character, it automatically looks for files that match the given pattern. In this case, it identified two such files. Then, it replaced the `*2*.csv` with the list of files, separated by spaces. In other the two commands:

```
ls *2*.csv  ls Allsource-201205.csv Tidal-EA-201205.csv
```

are exactly identical. The `ls` command cannot tell the difference between these two things.

Short Exercise

Do each of the following using a single `ls` command without navigating to a different directory.

1. List all of the files in /nocs that contain the letter a
2. List all of the files in /nocs that contain the letter a or the letter b
3. List all of the files in /nocs that contain the letter a AND the letter b

Tab Completion

Navigate to the home directory. Typing out directory names can waste a lot of time. When you start typing out the name of a directory, then hit the tab key, the shell will try to fill in the rest of the directory name. For example, enter:

```
cd S<tab>
```

The shell will fill in the rest of the directory name for {{page.bootcamp_slug}}. Now enter:

```
ls 3<tab><tab>
```

When you hit the first tab, nothing happens. The reason is that there are multiple directories in the home directory that start with 3. Thus, the shell does not know which one to fill in. When you hit tab again, the shell will list the possible choices.

Tab completion can also fill in the names of programs. For example, enter e<tab><tab>. You will see the name of every program that starts with an e. One of those is echo. If you enter ec<tab> you will see that tab completion works.

Command History

You can easily access previous commands. Hit the up arrow. Hit it again. You can step backwards through your command history. The down arrow takes your forwards in the command history.

^-C will cancel the command you are writing, and give you a fresh prompt.

^-R will do a reverse-search through your command history. This is very useful.

Which program?

Commands like ls, rm, echo, and cd are just ordinary programs on the computer. A program is just a file that you can *execute*. The program which tells you the location of a particular program. For example:

```
which ls
```

Will return "/bin/ls". Thus, we can see that ls is a program that sits inside of the /bin directory. Now enter:

```
which find
```

You will see that find is a program that sits inside of the /usr/bin directory.

So ... when we enter a program name, like ls, and hit enter, how does the shell know where to look for that program? How does it know to run /bin/ls when we enter ls. The answer is that when we enter a program name and hit enter, there are a few standard places that the shell automatically looks. If it can't find the program in any of those places, it will print an error saying, "command not found". Enter the command:

```
echo $PATH
```

This will print out the value of the PATH environment variable. More on environment variables later. Notice that a list of directories, separated by colon characters, is listed. These are the places the shell looks for programs to run. If your program is not in this list, then an error is printed. The shell ONLY checks in the places listed in the PATH environment variable.

Navigate to the nocs-shell directory and list the contents. You will notice that there is a program (executable file) called hello in this directory. Now, try to run the program by entering:

```
hello
```

You should get an error saying that hello cannot be found. That is because this directory is not in the PATH. You can run the hello program by entering:

```
./hello
```

Remember that . is a shortcut for the current working directory. This tells the shell to run the hello program which is located right here. So, you can run any program by entering the path to that program. You can run hello equally well by specifying:

```
/home/user/{{page.bootcamp_slug}}/lessons/nocs/nocs-shell/hello
```

Or by entering:

```
../nocs-shell/hello
```

When there are no `/` characters, the shell assumes you want to look in one of the default places for the program.

Examining Files

We now know how to switch directories, run programs, and look at the contents of directories, but how do we look at the contents of files?

The easiest way to examine a file is to just print out all of the contents using the program `cat`. Enter the following command:

```
cat appaloosa.txt
```

This prints out the contents of the `appaloosa.txt` file. If you enter:

```
cat appaloosa.txt appaloosa.txt
```

It will print out the contents of `appaloosa.txt` twice. `cat` just takes a list of file names and writes them out one after another (this is where the name comes from, `cat` is short for concatenate).

Short Exercises

1. Print out the contents of the `nocs-shell/dictionary.txt` file. What does this file contain?
2. Without changing directories, (you should still be in `nocs-shell`), use one short command to print the contents of all of the files in the `/home/user/{{page.bootcamp_slug}}/lessons/nocs/nocs-shell/data/nocs` directory.

---

cat is a terrific program, but when the file is really big, it can be annoying to use. The program, less, is useful for this case. Enter the following command:

```
less nocs-shell/dictionary.txt
```

less opens the file, and lets you navigate through it. The commands are identical to the man program. Use "space" to go forward and hit the "b" key to go backwards. The "g" key goes to the beginning of the file and "G" goes to the end. Finally, hit "q" to quit.

less also gives you a way of searching through files. Just hit the "/" key to begin a search. Enter the name of the word you would like to search for and hit enter. It will jump to the next location where that word is found. Try searching the dictionary.txt file for the word "cat". If you hit "/" then "enter", less will just repeat the previous search. less searches from the current location and works its way forward. If you are at the end of the file and search for the word "cat", less will not find it. You need to go to the beginning of the file and search.

Remember, the man program uses the same commands, so you can search documentation using "/" as well!

Short Exercise

Use the commands we've learned so far to figure out how to search in reverse while using less.

Redirection

Let's turn to the experimental oceanographic data on estimated abstractions from tidal waters and all other sources that we began with. This data is located in the nocs-shell/data/nocs directory. Lets print them all:

```
cat *.csv
```

Now enter the following command:

```
cat *.csv > ../all_data
```

This tells the shell to take the output from the cat *.csv command and dump it into a new file called ../all_data. To verify that this worked, examine the all_data file. If all_data had already existed, we would overwrite it. So the > character tells the shell to take the output from what ever is on the left and dump it into the file on the right. The >> characters do almost the same thing, except that they will append the output to the file if it already exists.

Information

Each subdirectory in nocs-shell/data other than nocs directory corresponds to a particular participant's hearing test data in the study. Navigate to the bert subdirectory in data. There are a bunch of text files that contain experimental data results.

Short Exercise

Use >>, to append the contents of all of the files, which contain the number 4 in the directory:

```
/home/user/{{page.bootcamp_slug}}/lessons/nocs/nocs-shell/data/gerdal
```

to the existing all_data file. Thus, when you are done all_data should contain all of the experiment oceanographic data from tidal waters plus all other sources and any experimental data file from gerdal that contains the number 4.

Creating, moving, copying, and removing

We've created a file called all_data using the redirection operator >. This file is critical - it's our analysis results - so we want to make copies so that the data is backed up. Lets copy the file using the cp command. The cp command backs up the file. Navigate to the data directory and enter:

```
cp all_data all_data_backup
```

Now all_data_backup has been created as a copy of all_data. We can move files around using the command mv. Enter this command:

```
mv all_data_backup /tmp/
```

This moves all_data_backup into the directory /tmp. The directory /tmp is a special directory that all users can write to. It is a temporary place for storing files. Data stored in /tmp is automatically deleted when the computer shuts down.

The mv command is also how you rename files. Since this file is so important, let's rename it:

```
mv all_data all_data_IMPORTANT
```

Now the file name has been changed to all_data_IMPORTANT. Let's delete the backup file now:

```
rm /tmp/all_data_backup
```

The mkdir command is used to create a directory. Just enter mkdir followed by a space, then the directory name.

Short Exercise

Do the following:

1. Rename the all_data_IMPORTANT file to all_data.
2. Create a directory in the data directory called foo
3. Then, copy the all_data file into foo

By default, rm, will NOT delete directories. You can tell rm to delete a directory using the -r option. Enter the following command:

```
rm -r foo
```

Count the words

The wc program (word count) counts the number of lines, words, and characters in one or more files. Make sure you are in the data directory, then enter the following command:

```
wc  nocs/* bert/* gerdal/*4*
```

For each of the files indicated, wc has printed a line with three numbers. The first is the number of lines in that file. The second is the number of words. Finally, the total number of characters is indicated. The final line contains this information summed over all of the files. Thus, there were 21912 characters in total.

Remember that the nocs/* and gerdal/*4* files were merged into the all_data file. So, we should see that all_data contains the same number of characters:

```
wc all_data
```

Every character in the file takes up one byte of disk space. Thus, the size of the file in bytes should be 14996. Let's confirm this:

```
ls -l all_data
```

Remember that ls -l prints out detailed information about a file and that the fifth column is the size of the file in bytes.

Short Exercise

Figure out how to get wc to print the length of the longest line in all_data.

The awesome power of the Pipe

Suppose I wanted to only see the total number of character, words, and lines across the files nocs/* and gerdal/*4*. I don't want to see the individual counts, just the total. Of course, I could just do:

```
wc all_data
```

Since this file is a concatenation of the smaller files. Sure, this works, but I had to create the all_data file to do this. Thus, I have wasted a precious 7062 bytes of hard disk space. We can do this *without* creating a temporary file, but first I have to show you two more commands: head and tail. These commands print the first few, or last few, lines of a file, respectively. Try them out on all_data:

```
head all_data tail all_data
```

The -n option to either of these commands can be used to print the first or last n lines of a file. To print the first/last line of the file use:

```
head -n 1 all_data tail -n 1 all_data
```

Let's turn back to the problem of printing only the total number of lines in a set of files without creating any temporary files. To do this, we want to tell the shell to take the output of the wc nocs/* gerdal/*4* and send it into the tail -n 1 command. The | character (called pipe) is used for this purpose. Enter the following command:

```
wc nocs/* gerdal/Data0559 | tail -n 1
```

This will print only the total number of lines, characters, and words across all of these files. What is happening here? Well, tail, like many command line programs will read from the *standard input* when it is not given any files to operate on. In this case, it will just sit there waiting for input. That input can come from the user's keyboard *or from another program*. Try this:

```
tail -n 2
```

Notice that your cursor just sits there blinking. Tail is waiting for data to come in. Now type:

```
French fries are good
```

then CONTROL+d. You should see the lines:

```
are good
```

printed back at you. The CONTROL+d keyboard shortcut inserts an *end-of-file* character. It is sort of the standard way of telling the program "I'm done entering data". The | character is replaces the data from the keyboard with data from another command. You can string all sorts of commands together using the pipe.

The philosophy behind these command line programs is that none of them really do anything all that impressive. BUT when you start chaining them together, you can do some really powerful things really efficiently. If you want to be proficient at using the shell, you must learn to become proficient with the pipe and redirection operators: |, >, >>.

A sorting example

Let's create a file with some words to sort for the next example. We want to create a file which contains the following oceanographic data parameters:

```
Temperature Salinity Velocity Pressure
```

To do this, we need a program which allows us to create text files. There are many such programs, the easiest one which is installed on almost all systems is called nano. Navigate to /tmp and enter the following command:

```
nano toBeSorted
```

Now enter the four names as shown above. When you are done, press CONTROL+O to write out the file. Press enter to use the file name toBeSorted. Then press CONTROL+x to exit nano.

When you are back to the command line, enter the command:

```
  sort toBeSorted
```

Notice that the names are now printed in alphabetical order.

Short Exercise

Use the `echo` command and the append operator, `>>`, to append your name to the file, then sort it and make a new file called Sorted.

Let's navigate back to `nocs-shell/data`. Enter the following command:

```
wc nocs/* | sort -k 3 -n
```

We are already familiar with what the first of these two commands does: it creates a list containing the number of characters, words, and lines in each file in the `nocs` directory. This list is then piped into the `sort` command, so that it can be sorted. Notice there are two options given to sort:

1. `-k 3`: Sort based on the third column
2. `-n`: Sort in numerical order as opposed to alphabetical order

Notice that the files are sorted by the number of characters.

Short Exercise

Use the `man` command to find out how to sort the output from `wc` in reverse order.

Short Exercise

Combine the `wc`, `sort`, `head` and `tail` commands so that only the `wc` information for the largest file is listed

Hint: To print the smallest file, use:

```
wc nocs/* | sort -k 3 -n | head -n 1
```

Short Exercise: Printing the smallest file seems pretty useful. We don't want to type out that long command often. Let's create a simple script, a simple program, to run this command. The program will look at all of the files in the current directory and print the information about the smallest one. Let's call the script smallest. We'll use nano to create this file. Navigate to the data directory, then:

```
nano smallest
```

Then enter the following text:

```
#!/bin/bash wc * | sort -k 3 -n | head -n 1
```

Now, cd into the bert directory and enter the command ../smallest. Notice that it says permission denied. This happens because we haven't told the shell that this is an executable file. If you do ls -l ../smallest, it will show you the permissions on the left of the listing.

Enter the following commands:

```
chmod a+x ../smallest ../smallest
```

The chmod command is used to modify the permissions of a file. This particular command modifies the file ../smallest by giving all users (notice the a) permission to execute (notice the x) the file. If you enter:

```
ls -l ../smallest
```

You will see that the file name is green and the permissions have changed. Congratulations, you just created your first shell script!

Searching files

You can search the contents of a file using the command grep. The grep program is very powerful and useful especially when combined with other commands by using the

pipe. Navigate to the bert directory. Every data file in this directory has a line which says "Range". The range represents the smallest frequency range that can be discriminated. Lets list all of the ranges from the tests that bert conducted:

```
grep Range *
```

Short Exercise

Create an executable script called smallestrange in the data directory, that is similar to the smallest script, but prints the file containing the file with the smallest Range. Use the commands grep, sort, and tail to do this.

Finding files

The find program can be used to find files based on arbitrary criteria. Navigate to the data directory and enter the following command:

```
find . -print
```

This prints the name of every file or directory, recursively, starting from the current directory. Let's exclude all of the directories:

```
find . -type f -print
```

This tells find to locate only files. Now try these commands:

```
find . -type f -name "*1*" find . -type f -name "*1*" -or -name "*2*" -print find . -type
f -name "*1*" -and -name "*2*" -print
```

The find command can acquire a list of files and perform some operation on each file. Try this command out:

```
find . -type f -exec grep Volume {} \;
```

This command finds every file starting from `.`. Then it searches each file for a line which contains the word "Volume". The `{}` refers to the name of each file. The trailing `\;` is used to terminate the command. This command is slow, because it is calling a new instance of `grep` for each item the `find` returns.

A faster way to do this is to use the `xargs` command:

```
find . -type f -print | xargs grep Volume
```

`find` generates a list of all the files we are interested in, then we pipe them to `xargs`. `xargs` takes the items given to it and passes them as arguments to `grep`. `xargs` generally only creates a single instance of `grep` (or whatever program it is running).

Short Exercise

Navigate to the `data` directory. Use one `find` command to perform each of the operations listed below (except number 2, which does not require a `find` command):

1. Find any file whose name is "NOTES" within `data` and delete it
2. Create a new directory called `cleaneddata`
3. Move all of the files within `data` to the `cleaneddata` directory
4. Rename all of the files to ensure that they end in `.txt` (note: it is ok for the file name to end in `.txt.txt`)

Hint: If you make a mistake and need to start over just do the following:

1. Navigate to the `nocs-shell` directory
2. Delete the `data` directory
3. Enter the command: `git checkout -- data` You should see that the data directory has reappeared in its original state

BONUS

Redo exercise 4, except rename only the files which do not already end in .txt. You will have to use the man command to figure out how to search for files which do not match a certain name.

Bonus:

backtick, xargs: Example find all files with certain text

alias -> rm -i

variables -> use a path example

.bashrc

du

ln

ssh and scp

regular expressions

permissions

chaining commands together