An Introduction to the UNIX Shell

Paul La Plante

June 1, 2020

1 What is the shell?

The shell, or command line interface (CLI), is one of the most fundamental ways to interact with a computer. Rather than using a mouse to click on icons or windows, the shell relies entirely on typing commands. At a low level, the shell allows the user to access the hardware at the heart of the machine. Part of what makes the shell so powerful is its ability to run other commands, which in turn can accomplish complicated tasks or control things automatically. By the end of this tutorial, you will be able to navigate your system using the command line, as well as run other programs. You'll also be well on your way to understanding how to write your own programs for solving problems. Let's get started!

These notes are based on the lessons presented by the Software Carpentry website, `http://swcarpentry.github.io/shell-novice/`. There are many other valuable lessons there, beyond just the shell. Definitely check it out!

2 Navigating using the shell

An important first question when learning to use the shell is "How do I get around my filesystem?" In some sense, the command line gives you a text version of the directory structure that you might be used to navigating graphically, such as with Windows Explorer or macOS Finder.

2.1 UNIX Filesystem Structure

UNIX (and its descendents) uses a hierarchical file system, which means that there is a root directory (denoted as `/`), with files and subdirectories that are located within that directory. Those subdirectories can in turn contain files and sub-subdirectories of their own, to help organize the contents of the filesystem. Each file and directory "knows" which directory it is located in, so that you can eventually move up the directory structure (sometimes called a "tree") and end up back at the root directory `/`.

One of the benefits of the hierarchical structure is that each file or directory has a unique name associated with it. Subdirectories are denoted by a `/` character between its name and

that of its parent directory. For instance, your "home" directory, which is where Ubuntu defaults to putting your personal files and downloaded files, is called "champ1", which is located in the `home` directory, which is located in the root directory `/`. The folder's location in the filesystem can be represented as:

```
/home/champ1
```

This directory is frequently used when writing commands, and so the tilde character `~` can be used to substitute for it.

2.2 Our first commands: `ls`, `cd`, and `pwd`

2.2.1 `ls`

Now that we know how the filesystem is laid out, we can begin using the command line to move around. Let's begin by opening a new terminal. You should see something that looks like this:

```
$
```

This dollar sign is known as the "command prompt". This symbol means the shell is listening for your input, and is ready to run a command.

The first thing we want to do is to see what is in the current directory. Let's find out! The command for this is `ls` (the "listing" of the directory's contents). To run the command, we type the characters `ls`, followed by the Enter key. You should see something like this:

```
$ ls
Desktop Documents Downloads Pictures Videos pic.png
$
```

The first line (`ls`) is the command we entered, and the next line (Documents, Downloads, etc.) is the output of that command. On the following line is the command prompt again, meaning the shell has finished executing the command we asked it to run and is listening for the next command.

Commands will often have different options available for us to use, which are usually entered by entering a dash and a character after the command name. Let's look at `ls` as an example. When listing directory contents, we can have the computer tell us which entries are directories with a trailing slash by using the `-F` option. Let's try it:

```
$ ls -F
Desktop/ Documents/ Downloads/ Pictures/ Videos/ pic.png
```

Now we can see that every entry in our home directory is another directory, except for one. We will soon learn how to move between directories using another command.

As a side note, you can see many of a command's options by using the `--help` option. Alternatively, you can look at a command's `man`ual page by running `man <command>`, *e.g.*,

`man ls`. This will bring up a separate environment, where you use the space bar to move forward and `b` to move back. You can quit using the `q` key. The man page has more information than you ever cared to know about a specific command, so if you have specific questions, it's a great place to start.

We can also provide "arguments" to commands, which change how they behave. For the `ls` command, if we give it an argument, we will see a list of that directory's contents instead of the one we're currently in. Let's try it with the `Desktop` folder:

```
$ ls -F Desktop
data-shell/
```

We can now use `ls` to look at the contents of this sub-directory:

```
$ ls -F Desktop/data-shell
creatures/        molecules/          notes.txt         solar.pdf
data/             north-pacific-gyre/ pizza.cfg         writing/
```

We can see that there are more directories and files in this directory. Now we're going to learn how to change our directory so we're in this one.

2.2.2 `cd`

How do we move between directories? We need a new command, called `cd` (for change directory). Let's move to the `data-shell` directory we saw before:

```
$ ls
Desktop Documents Downloads Pictures Videos pic.png
$ cd Desktop
$ cd data-shell
$ cd data
$ ls -F
amino-acids.txt    elements/    pdb/         salmon.txt
animals.txt        morse.txt    planets.txt  sunspot.txt
```

The `cd` command didn't produce any output, but if we run `ls` after the `cd` command, we see that we have several new entries.

2.2.3 Aside: `pwd`

When we're moving around the file structure, it's easy to get lost and forget where you are. The `pwd` command will print the working directory. Let's try it:

```
$ pwd
/home/champ1/Desktop/data-shell/data
```

We now know where we are, which is very helpful for when we've moved around and forgotten where exactly we're located. Now, back to `cd`.

2.2.4 Back to `cd`

So now that we've learned how to move down in the directory hierarchy, let's find out how to move back up. One thing you might try is `cd data-shell`, the name of the parent directory. Let's see what happens:

```
$ cd data-shell
cd: no such file or directory: data-shell
$ pwd
/home/champ1/Desktop/data-shell/data
```

`cd` gave us an error message, and we didn't actually move anywhere! What happened? The command `cd` is only able to move to directories that are contained in the current one. Fortunately in UNIX, every folder "knows" what directory it's in. Let's see how:

```
$ cd ..
$ pwd
/home/champ1/Desktop/data-shell
```

`..` means "this directory's parent directory". But wait! We said that `cd` can only move to directories contained in the current one! We can see that it actually *is* in the current directory, it's just "hidden". Let's try `ls` with the `-a` option:

```
$ ls -F -a
./              creatures/          notes.txt
../             data/               pizza.cfg
.bash_profile   molecules/          solar.pdf
Desktop/        north-pacific-gyre/ writing/
```

The "`.`" and "`..`" entries are special, and are found in every directory. "`.`" means "the current directory", and we'll see some uses for it soon. We also see the "`..`" entry, the parent directory. We also see a new file we didn't before, `.bash_profile`. By default, `ls` doesn't show any files or directories that begin with a "`.`" character. These are usually configuration files the user does not usually have reason to modify, so they are hidden to prevent cluttered directories.

As a couple of last points on `cd`, let's see what happens if we run `cd` without any commands:

```
$ cd
$ pwd
/home/champ1
```

We were brougt back to our home directory. We can also use the `~` character to represent our home directory:

```
$ cd ~/Desktop/data-shell
$ ls -F
creatures/        molecules/          notes.txt         solar.pdf
data/             north-pacific-gyre/ pizza.cfg         writing/
```

We're back in the `data-shell` directory.

2.2.5 Aside: Absolute vs. Relative Paths

When we want to specify the path to a file or directory in UNIX, there are two options: absolute paths and relative paths. Absolute paths contain the entire path, starting from the root directory `/`. These paths work no matter which directory we're located in, since they tell the filesystem how to get somewhere starting from the root directory.

Conversely, we've seen that we can reference a directory's parent by using the `..` alias. Let's suppose that we're in `~/Documents`. If we wanted to get to `~/Desktop`, we could use the command:

```
$ cd /home/champ1/Desktop
```

OR

```
$ cd ../Desktop
```

In both cases, we'd end up in the same folder.

2.2.6 Aside: tab-completion

Are you tired of typing out the entire name of a file or directory yet? Or making typos? (This is a recurring problem for me...) Enter tab-completion, which is one of the best features of the shell.

Tab-completion works by typing the first few letters of a command or argument, and then hitting the `<TAB>` key. If the shell knows what you're trying to say, it will automatically fill in the rest for you! If it doesn't know exactly what you want (like if there are several different things that match the first few letters you've typed), then hitting `<TAB>` a second time will print out to the screen all of the possible completions for your partial command or argument. So, if you type `cd ~/Desk<TAB>`, the shell will auto-complete to `cd ~/Desktop`. On the other hand, typing `cd ~/D<TAB>` will not auto-complete, since this could turn into `~/Desktop`, `~/Documents`, or `~/Downloads`. Hitting `<TAB>` a second time will show you these options, so you can see for yourself why the shell isn't completing the command.

Congrats! You now know how to move around to different directories. This an important first step on the road to mastering the shell.

3 Files and Directories

3.1 Making New Directories

It's helpful to know how to move around existing directories, but we also want to be able to make our own new ones. For this, we have the `mkdir` command, which is short for "make directory". Let's do that now:

```
$ cd ~/Desktop/data-shell
$ ls -F
creatures/        molecules/          notes.txt         solar.pdf
data/             north-pacific-gyre/ pizza.cfg         writing/
$ mkdir thesis
$ ls -F
creatures/   north-pacific-gyre/   thesis/
data/        notes.txt             writing/
Desktop/     pizza.cfg
molecules/   solar.pdf
```

We see that we have a new `thesis` directory. Great!

3.1.1 Naming Files and Directories, a.k.a., Spaces are Evil

When we're naming our files and directories, there are some important rules that we should follow:

1. Don't use spaces in names. Use `_` (underscore) or `-` (dash) instead. Whitespace in directory names is frequently garbled by the shell unless you're careful, since it assumes a space means "move on to the next argument".

2. Don't begin names with `-`, since the shell interprets leading dashes as command arguments.

3. Try to use only letters, numbers, `.` (period), `_` (underscore), and `-` (dash). We're going to learn soon that most punctuation marks have special meaning in the shell.

4. *Don't use spaces in names.*

3.2 Making New Files

Now that we've made a new directory, we want to put something in it. We're going to start a new text file. To do that, we're going to use a program called `nano`. Let's make a new file called `draft.txt`:

```
$ cd thesis
$ nano draft.txt
```

This opens up `nano`, which allows you to write plain text files. We're going to use it (or another text editor) to write our programs in later lessons, so it's good to get familiar with it. You can type the contents of the file into `nano` directly. You'll notice at the bottom of the screen, there are commands listed. The combination of `^O`, for instance, means to hold down the `Ctrl` key while pressing `O`. Let's write some text, save it with `^O`, and exit with `^X`.

Let's look at our directory now:

```
$ ls
draft.txt
```

We didn't like that draft, so we're going to start over and try again. We get rid of the current file using the `rm` (for "remove") command:

```
$ rm draft.txt
$ ls
```

There's no output from `ls` anymore, because we've deleted the only file that was in this directory. Let's get rid of the directory too:

```
$ cd ..
$ rm thesis
rm: cannot remove 'thesis': Is a directory
```

The `rm` command by default cannot remove directories. There is a command `rmdir` to do that, or we can pass an option to `rm`:

```
$ rm -r thesis
```

This will get rid of the directory, and all of its contents. (The `-r` option instructs `rm` to recurse into all subdirectories.)

3.2.1 Aside: Always Have the Safety On

One *very* important aspect of `rm` is that once deleted, the files are gone forever. There is no "trash can" or "recycle bin" on the command line: once a file has been `rm`'d, there is no way to get it back. To get around this, we can use the `rm -i` command, which will interactively ask you about every file, and whether you really want to delete it. This gives you an extra layer of security, and keeps you from accidentally deleting all of your code you worked so hard to write.

3.3 `cp` and `mv`: Moving Things Around

The `cp` command is very straightforward: it copies files (or directories) from one place to another. The general syntax is `cp existing_file new_file`. The contents of `existing_file` will be copied to `new_file` as an exact copy. This gives us 2 versions of the file, and can be useful for making a backup copy of code or data files.

The `mv` command is also straightforward: it uses the same syntax as the `cp` command, but deletes the original copy. So, if you run `mv existing_file new_file`, you'll notice that there's only one file, `new_file`, when you're done. `mv` can be used to move files from one directory to another, or to rename files and directories (*e.g.*, using `mv old_name new_name`).

Let's use this in practice to get one of the data directories off of the desktop folder, and into our `Documents` folder:

```
$ cd ~/Desktop/data-shell
$ mv north-pacific-gyre ~/Documents
```

We also could have given a relative path, `mv north-pacific-gyre ../../Documents`. Note that `mv` will not produce output unless the `-v` option is given to it. This has moved the data from where it was to a new location. All of the files contained in this directory are still inside of it, just at a new point in the filesystem.

4 Pipes and Filters

One of the most powerful features of UNIX is that you can very easily use the output of one command as the input for another. This allows us to chain together commands to do powerful tasks with just a few keystrokes. Let's see how these behave.

4.1 Wildcards

We're going to use a new command, `wc`, which will give us a word count of a file that we pass as an argument. Let's go to the `molecules` folder:

```
$ cd ~/Desktop/data-shell/molecules
$ ls
cubane.pdb   ethane.pdb   methane.pdb
octane.pdb   pentane.pdb  propane.pdb
```

4.1.1 The * wildcard

We have six files here. Rather than run the `wc` command on each individually, we're going to use what's known as a "wildcard" character. The `*` symbol tells the shell to match zero or more characters. For instance, let's use it to run `wc` on all files that end in `.pdb`:

```
$ wc *.pdb
 20  156 1158 cubane.pdb
 12   84  622 ethane.pdb
  9   57  422 methane.pdb
 30  246 1828 octane.pdb
 21  165 1226 pentane.pdb
 15  111  825 propane.pdb
107  819 6081 total
```

By reading the `man` page of `wc`, we would see that the three columns are the number of lines, words, and characters contained in each file. The total of all files inspected is the bottom row.

4.1.2 The `?` wildcard

There is another commonly used wildcard, the `?` character, which is similar to the `*` character. Instead of matching any number of characters, the `?` character matches exactly one character. So if we ran `wc *ethane.pdb`, we'd get word counts for both `ethane.pdb` and `methane.pdb`. On the other hand, if we ran `wc ?ethane.pdb`, we would only get `methane.pdb`, since the `?` character must match at least one character.

4.2 Redirecting Output

Let's return to the `wc` command. The option `-l` will give us just the number of lines in a file (the first column of the full `wc` command):

```
$ wc -l *.pdb
 20  cubane.pdb
 12  ethane.pdb
  9  methane.pdb
 30  octane.pdb
 21  pentane.pdb
 15  propane.pdb
107  total
```

This output is manageable, since there are only six files. What if we had thousands? We can redirect the output of the `wc` command to a file, so that we can read it more easily. To do this, we use the `>` (greater than) symbol:

```
$ wc -l *.pdb > lengths.txt
```

Note that there's no output from this command, since it was sent to `lengths.txt`. Let's look at our directory contents again:

```
$ ls
cubane.pdb   ethane.pdb   lengths.txt   methane.pdb
octane.pdb   pentane.pdb  propane.pdb
```

We have a new file, `lengths.txt`, that has shown up. Now let's take a look at what it contains.

4.2.1 `cat` and `less`: Seeing File Contents

To see the contents of a file, we can use `cat` or `less`. `cat` will concatenate the file to the screen, and prints the whole file. `less` will show you `less` of the file—only one screen's worth—and is much easier for navigating large files. Like `man` pages, you can move through the `less` screen by using space bar to go forward a page, `b` to go back, and `q` to quit. Since we're looking at just a small file, we're going to use `cat`.

In general, though, `less` is more!

Okay, now let's look at the contents of `lengths.txt`:

```
$ cat lengths.txt
 20  cubane.pdb
 12  ethane.pdb
  9  methane.pdb
 30  octane.pdb
 21  pentane.pdb
 15  propane.pdb
107  total
```

Notice that this is the same as the output from `wc -l *.pdb` from before, without the redirect.

4.3 Sorting Output

Oftentimes, we want to be able to sort our output. Let's find out how to sort our `lengths.txt` file to figure out the longest file:

```
$ sort -n lengths.txt
  9  methane.pdb
 12  ethane.pdb
 15  propane.pdb
 20  cubane.pdb
 21  pentane.pdb
 30  octane.pdb
107  total
```

We can see that `methane.pdb` is the shortest file, and `octane.pdb` is the longest one. We can make a new file that has the lengths sorted. We also want to use the `head` command, to see just the top line of the resulting file:

```
$ sort -n lengths.txt > sorted-lengths.txt
$ head -n 1 sorted-lengths.txt
  9 methane.pdb
```

Giving the `head` command the option `-n 1` means that we see only the first line of the file.

4.4 Pipes

Wouldn't it be convenient if there were a way to avoid making a temporary file? We can accomplish this by using the pipe character `|` bewteen two commands to tell the shell "the output of the first command should become the input for the second command." We can get the same output result as above, but without having to make the `sorted-lengths.txt` file:

```
$ rm sorted-lengths.txt
$ sort -n lengths.txt | head -n 1
  9 methane.pdb
```

If we were to `ls` in the directory, we would see that we didn't create a `sorted-lengths.txt` file this time. We can even chain multiple commands together, so we wouldn't even need a `lenghts.txt` file!

```
$ wc -l *.pdb | sort -n | head -n 1
  9 methane.pdb
```

Ta-da! No temporary files!

This paradigm is what makes UNIX so powerful and so successful: make a tool, that does one thing *really* well, and then use pipes to redirect output between tools, which act as filters to massage the original output into our desired format.

4.5 Putting it all together

Now, let's use what we learned on a small scale to look at (a facsimile of) real data. We're gong to go to the `north-pacific-gyre` folder:

```
$ cd ~/Documents/north-pacific-gyre
$ wc -l *.txt | sort -n | head -n 5
 240 NENE02018B.txt
 300 NENE01729A.txt
 300 NENE01729B.txt
 300 NENE01736A.txt
 300 NENE01751A.txt
```

As we saw above, this will give us the number of lines of all files that end in `.txt`, sort them, and then give us the top 5 entries (in this case, the 5 shortest files). We can see that one file is significantly shorter than the rest of them.

Let's also take a look at the 5 longest ones. This is done with the `tail -n 5` command:

```
$ wc -l *.txt | sort -n | tail -n 5
 300 NENE02040B.txt
 300 NENE02040Z.txt
 300 NENE02043A.txt
 300 NENE02043B.txt
5040 total
```

We've gotten some glimpses into why the command line is so powerful, and we've only used a few commands!

5 Loops

Loops are one of the most basic constructs in all of programming. Essentially, a loop says "I want to repeat the same set of actions many times, with only small changes each time." As control structures, they are clear to both humans and machines, and are very efficient at automating large tasks. Automation is one of the major benefits of programming in the first place, so we're going to take our first step toward how to do it in the shell.

5.1 For Loops

A "for loop" is a construct where we perform some task *for* a certain number of pre-defined iterations. To see this in action, let's go to another directory. Our first task is to make a backup of each of these files. Let's use `cp` for this:

```
$ cd ~/Desktop/data-shell/creatures
$ cp *.dat original-*.dat
cp: target 'original-*.dat' is not a directory
```

What happened? When we give `cp` multiple arguments, it expects the final one to be a directory to copy all of the files to. In this case, the first wildcard is expanded, so the full command reads `cp basilisk.dat unicorn.dat original-*.dat`. Since there is no directory with that name, `cp` gives us an error.

We can use a loop to accomplish this for us. Enter the following commands:

```
$ for filename in *.dat
> do
>    cp $filename original-$filename
> done
```

Note that the prompt has changed from the dollar sign to the "greater than" sign. This means that the shell is waiting for us to finish the for loop, and will change back once the loop has been completed.

Now we have achieved the original intent: we've made a backup of each of our files. Implicitly, in this construction, we have told the shell that it should do the same thing to two different files: for each file in our list, we want to make a copy, with `original-` prepended to the filename.

When we're building a loop, sometimes it's helpful to print the command you *actually* wrote, and see if it's the same as the one that you *wanted* to write. To do this, we use the `echo` command:

```
$ for filename in *.dat
> do
>    echo "cp $filename original-$filename"
> done
cp basilisk.dat original-basilisk.dat
cp unicorn.dat original-unicorn.dat
```

Now we can see that the commands do what we expect them to do, and we can remove the `echo` and quotes to actually run the command. We've already done this, so we won't do it again. Just know that echoing the body of the loop is helpful for debugging potential issues.

6 Scripts

We've finally arrived at the teleological conclusion of the shell: scripts to do the business of repetitive tasks for us. You might have noticed that writing for loops by hand is repetitive and error-prone. It's also difficult to change a few small things and run the same commands again. We will now learn about writing scripts.

To start with, we're going to use `nano` again. Let's write a script that will take lines 11-15 of the `octane.pdb` files. Let's go back to the `molecules` folder.

```
$ cd ~/Desktop/data-shell/molecules
$ nano middle.sh
```

This makes a new file called `middle.sh`. In nano, let's write the following line:

```
head -n 15 octane.pdb | tail -n 5
```

We can save with `^O` and exit with `^X`. Let's run it now, using the `bash` command:

```
$ bash middle.sh
ATOM      9  H           1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H           1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H           1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H           1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H           1      -3.172  -1.337   0.206  1.00  0.00
```

Excellent! Now we can make changes to the shell script, and rerun it easily. It's not that flexible though: we can only run this on `octance.pdb`, and it gives us lines 11-15. Let's make some modifications to make it more flexible.

The flexibility can be achieved by using "positional arguments" which are represented by numbers. When we run a command, the variable `$1` represents the first argument passed in, `$2` is the second one, and so on. Let's open the file with `nano` again, and change the line to read:

```
head -n 15 "$1" | tail -n 5
```

We've protected the first argument by placing it in double quotes, which will allow for filenames with evil spaces to be passed in without generating an error. Now let's run `middle.sh` again, with a different file:

```
$ bash middle.sh pentane.pdb
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

Great! We've added more flexibility. Let's make it even more flexible, and also allow the user to specify the numbers that get passed into `head` and `tail`:

```
head -n "$2" "$1" | tail -n "$3"
```

We can recover the same behavior as above using a new command:

```
$ bash middle.sh pentane.pdb 15 5
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

At the same time, we can look at lines 16-20 instead:

```
$ bash middle.sh pentane.pdb 20 5
ATOM     14  H           1      -1.259   1.420   0.112  1.00  0.00
ATOM     15  H           1      -2.608  -0.407   1.130  1.00  0.00
ATOM     16  H           1      -2.540  -1.303  -0.404  1.00  0.00
ATOM     17  H           1      -3.393   0.254  -0.321  1.00  0.00
TER      18              1
```

As we're going to learn, it's very helpful to document what was on your mind while you were writing the script, to help other programmers (or yourself!) to more easily understand what a program is doing. To do this, we'll add a few lines to the top of the script that start with the `#` character, which causes the shell to ignore the contents of that line. We'll open `nano` again, and run it:

```
# Select lines from the middle of a file.
# Usage: bash middle.sh filename end_line num_lines
head -n "$2" "$1" | tail -n "$3"
```

Great! We've successfully written our first shell script! You are now ready to go out and conquer the world by automating all of your tasks and running complicated code with a few keystrokes!

7 ssh and scp: How to Work on a Supercomputer

As you're going to see in the course of this bootcamp, most of the work that we do is too much for a single laptop or desktop computer to handle. Many times, we log into a different computer, typically a supercomputer, to help us with our most computationally challenging tasks. This involves getting an account on the machine, which has a username and password. Then, logging in is as simple as running a few commands on the command line.

7.1 ssh: Logging In

The most common tool for accessing your account on a different machine is `ssh`, which is a **s**ecure **sh**ell. In addition to your username and password, you'll need to know the remote computer's *hostname*, which is how you'll tell your machine which computer to connect to. The general structure for a command is:

```
$ ssh <user>@<hostname>
```

At this point, the remote machine will ask for your password.

Let's suppose that your username is `champ1`, and the computer we'd like to connect to is the NRAO computer. For this machine, the hostname is `login.aoc.nrao.edu`. So, the `ssh` command is:

```
$ ssh champ1@login.aoc.nrao.edu
champ1@login.aoc.nrao.edu's password:
```

At this point, you would enter your password. Note that the cursor does not move, to keep people looking over your shoulder from knowing how long your password is. Once you've entered it, the machine will grant you access, and give you a new command prompt. This time, you're running a shell remotely on the new machine. Exciting! From here, you can access data that's on the other machine, as well as run programs. We'll talk more about how to do these things in future lessons.

7.2 scp: Bringing Data Home

Let's assume that you've written a great script, `do_cool_science.sh`, and generated new data or a plot that you want to have on your personal machine. `ssh` is only a way to execute commands remotely, not a way to move files. Fortunately, there's a command `scp` to **s**ecurely `cp` files over the network. It uses the same security as `ssh` to make sure other people can't eavesdrop on your connection, and is very simple for moving files to or from the remote machine.

From your machine, you can either "push" files to the remote computer, or "pull" them from the remote machine to yours. Unless you've set up your personal computer's connection in a special way, it's much harder to push and pull data from the remote machine to your personal one. So you should always run the `scp` command from your personal machine.

The typical `scp` command for pushing to the remote machine is:

```
$ scp <local_data> <user>@<hostname>:<remote_destination_folder>
```

For instance, let's say that I have a file on my machine called `awesome_data.txt`, and on the remote machine I want to copy it to `~/Documents/data_backup`. The command I would run is:

```
$ scp awesome_data.txt champ1@login.aoc.nrao.edu:~/Documents/data_backup
champ1@login.aoc.nrao.edu's password:
```

After you enter your password, `scp` will begin transferring the file. A copy is going to end up on `login.aoc.nrao.edu`, in the `~/Documents/data_backup` folder, just like we wanted. `scp` will show you the status of the data transfer while it's happening, with information like how much of the file has been transferred so far, what the current speed is, and how long the transfer has been going on for. Once it's done, it will display average values for the whole transfer.

For pulling data from the super computer to your local machine, we use a similar syntax:

```
$ scp champ1@login.aoc.nrao.edu:~/Documents/data_backup/old_data.txt .
```

Once again, it will ask for our password. Note that we've used the shortcut `.` to represent the current directory. This means that a copy of the file `old_data.txt` will end up in the directory we're currently in.

8 `git`: Copying Code The Easy Way

`git` is a powerful command-line tool that lets you keep track of your code and share it with others, called a version control system (VCS). `git` can seem tricky, especially if you're learning it for the first time, and later lessons in the camp will explore it in more detail. But for now, the only thing we'll worry about is making a copy of an existing codebase, which we call *cloning a repository*. Many of the software repos we use are hosted on GitHub.

8.1 Quick aside: `git` vs. GitHub

There is a difference between `git`, the software VCS tool, and GitHub, which is a website that hosts software repositories (primarily managed with `git`). It's similar to how Python is a programming language, and Anaconda is a website that builds friendly tools and environments around the underlying Python. GitHub is a nice visual interface to the repository at different stages in history. It also lets you make Issues about the existing code, and Pull Requests to make changes or updates. But at its core, `git` is just a (very powerful) command line tool, and GitHub is one particular place that people host a lot of software (managed via `git`).

8.2 Cloning a `git` repo

When we want to make a local copy on our machine of a remote repo, we use the `git clone` command. This tells `git` where to find the repo, and instructs it to make a local copy. You can create and manage repos locally as well, but most often we'll use a hosting service like GitHub to track our repository history.

Let's make a new folder for our git repo. Let's call it `hera_software`:

```
$ mkdir hera_software
$ cd hera_software
$ git clone https://github.com/RadioAstronomySoftwareGroup/pyuvdata
Cloning into 'pyuvdata'...
remote: Enumerating objects: 304, done.
remote: Counting objects: 100% (304/304), done.
remote: Compressing objects: 100% (187/187), done.
remote: Total 19326 (delta 204), reused 161 (delta 117), pack-reused 19022
Receiving objects: 100% (19326/19326), 91.87 MiB | 6.29 MiB/s, done.
Resolving deltas: 100% (14473/14473), done.
Updating files: 100% (760/760), done.
$ cd pyuvdata
$ ls
CHANGELOG.md        README.md         paper.bib       scripts
CODE_OF_CONDUCT.md  ci                paper.md        setup.cfg
LICENSE             docs              pyproject.toml  setup.py
MANIFEST.in         environment.yaml  pyuvdata
```

So to recap: we made a new folder, called `hera_software`, `cd`'d into it, and then ran `git clone <repo address>`. As a part of this command, `git` made a new folder that matched the name of the repo, and downloaded a full copy of its current state. In addition, it also has all of the historical information about how the code has evolved. There are lots of neat things you can do with `git` in terms of how to track changes you make to code, and go back in time to a previous version of the code. There will be another lesson later about how to use `git` in more detail, but `git clone` should be enough to get started.

9 The End

Congratulations! You've taken your first steps toward mastering using the shell, one of the most powerful modern tools for science research and discovery. You're well on your way to becoming a data scientist, and gaining new insight on problems you wouldn't have otherwise. Don't be afraid to ask for help if you have problems, and the Internet is your friend if you have any questions!