More Command Line Tools

Previously, we learned how to navigate a file system from the command line, create files, view their contents, and move them around. Now we will look at some more useful command line tools.

Searching for and Within Files

Open a terminal or git bash window and navigate to the desktop.

```
cd ~/Desktop
```

You will notice that I used the `~` directory. This is not quite a local or an absolute path; it is customized for the end user and is a link to the end user's home directory. This means nothing more than where the end user would be able to find `/Desktop` or `/Downloads`; it is the main directory that appears when you open the command line.

find

Once we are there, a common exercise will be trying to find files. There are only two ways that you will find files: search for them by name or search for them by contents.

Searching by name is done with the `find` command. For example:

```
#### From Our Desktop
find . -name "*file*"
```

should print out

```
./test_creation/my_file.txt
./test_creation/second_file.txt
```

You will notice my use of stars in the above example. This is a nice piece of functionality. A `*` is a wild card and could be any character. You can get very expressive with this miniature language called regular expressions, and we will be getting into that later in this course.

In addition to the `-name` parameter that we passed, there are quite a few different parameters that you can pass into the `find` command; you can specify whether it should search deeply first or search the top levels first, when the file was created, and more. Remember you can use `man find` to see all the options.

grep

Another command you should be aware of is `grep`. This is a command that I use frequently because, rather than finding files by name, it searches by the contents of the file. We can use it to specify a specific file or a whole group of files.

For example

```
grep "Hello" test_creation/my_file.txt
```

would return

```
/test_creation/my_file.txt:Hello From the Command Line
```

Should "From the Command Line" above be on a separate line? If so, it should be all lowercase with a period at the end.

If I did not know which file it was in, I could use the `-r` flag to make a recursive search of a directory.

```
grep -r "second" .
```

would recursively search within my current directory (desktop). I could also just search within our test directory.

```
grep -r "second" test_creation/
```

There are many flags that you can pass to grep; this is just a sample. Now we are going to move on to a more destructive section.

The Danger Zone: Removing Files

This section is called the danger zone for a reason. You can do **serious** damage to your system with these commands. Likely hundreds of thousands of dollars of damage have been done with fewer than 10 keystrokes. This means that you *really* need to think about what you are doing before executing these commands. They can destroy directories and files.

Make sure that you are in our test_creation folder.*"folder" should not be italic."

```
cd Desktop/test_creation
```

Now let's make a new directory and some files to test.

```
mkdir test_destruction
touch test_destruction/1.txt
touch test_destruction/2.txt
touch test_destruction/3.txt
```

Now we are going to start removing files from these directories. We have created a directory and three files. Let's delete the last one.

```
rm test_destruction/3.txt
ls test_destruction
```

```
test_destruction/1.txt
test_destruction/2.txt
```

You will see that it deleted the last file. However this is not an instance where you can simply look in a recycle bin to find it again. This file is gone. It is not in a recovery bin. Only doing some forensic extraction could get it back–something that you do not want to do. Be sure to triple check that you are deleting the right file when you use the `rm` command.

Do you remember when we used the `*` in `find`? That wild card works with all commands. Do you want to delete everything in test destruction?

```
rm test_destruction/*
ls test_destuction
```

You will likely get a prompt before you can do this, but once it is done, you should see the power of this tool. With seven keystrokes, you can destroy everything on your computer–literally everything. We are not responsible for anything that you delete that you do not mean to delete, so use this tool wisely because you can do serious damage if you are not careful.

Now let's use `rmdir` to remove our test destruction folder. You will notice that

```
rm test_destruction
```

will fail. This is because rm is intended for files, not folders (although it can be use for such). You will need `rmdir` to remove the directory.

```
rmdir test_destruction
ls
```

should show you that the folder is destroyed. Again, be careful with what you delete using this tool. I use it because I am comfortable with it, but make sure that you triple check every time you think you should delete something.

Connecting Commands: Piping and Passing Output

In our introduction we covered a lot of these, but I wanted to show you one last tip. *What does "these" refer to? "Commands" or something else?* This is the `|` character. Try this from your desktop.

```
ls | grep "test"
```

We are piping the output from one command to another. We are taking the output of `ls` and outputting in into the `grep` command. If you have run this correctly, you should see the `test_creation` directory listed in this output.

Something else you can do is something like this.

```
echo "Does this contain a ?" | grep -o "?"
```

This will output one `?` because the `-o` flag signifies that it should only return matching characters. If you remove the `-o`, you will see the entire `echo` statement. This piping capability is very powerful and sits at the base of the `>` and `>>` character combinations that we covered in previous lessons.

Another combination that you will come across is `<`, which basically allows you to pipe a file or command another way.

```
grep "echo" < test_creation/my_file.txt
```

will basically force that file into `grep`. While this is used less, it is not uncommon to see it come up in other people's code.

Conclusion

This was a quick introduction and was not intended to be rigorous; it is meant to give you the basics of what you are going to need to know to succeed as a data scientist and programmer. This is by no means an introduction to everything that you can do at the command line. In fact, the command line has its own complete programming language called Bash. *When I was doing a search to see if "Bash" should be capped, one source said that it is not a programming language but a scripting language. Please verify.*

Zed Shaw, an amazing teacher, has an excellent cheat sheet that is worth looking at. It will show you a lot more commands that you should become familiar with at the command line. You can find that cheat sheet here: http://cli.learncodethehardway.org/bash_cheat_sheet.pdf.

One of the reasons that you will find this so useful is once you start getting a sense for the commands you write a lot. For example, I find myself looking for files at the command line a fair amount, so I wrote a function `ff`. *Text is missing from first sentence. It starts out "one of the reasons" but doesn't provide a reason.*

```
function ff { find . -name $1; }
```

This function makes it extremely easy to search for files. Just type `ff "fname"`, and it searches within the current directory.

When you know what the file is like, you can just write `ff "*file*"` and it will find everything that contains `file` in the file name. You should write or use these only when you need them, but this shows you the power of the shell.