Introduction to the Command Line or Shell

Before we start learning Python, it will be useful for us to begin with some command line practice.

The command line is an interface by which you can interact with the computer using commands rather than clicking and navigating with your mouse. While it may be uncomfortable and appears as if you are hacking a computer, you are simply issuing commands to the computer just as you would with a click.

Remember that there are several different languages used at the command line. We will be learning bash script, which is a standard language on Unix/Linux systems, like many of the servers you will need to control in this program, and on Macs.

Basic Navigation and Layout

Let's get started by opening up your terminal on a Mac or your git bash shell on Windows. Once you are in, you should see a screen start with `$`.

Now let's type

```
pwd
```

This stands for *print working directory*. A directory is like a folder and is your current location in the file system.

Now let's type

```
ls
```

This lists all of the files and folders in the directory. You will likely see a lot when you first run it. This is your computer's file system, where all programs and files and folders are stored on your computer.

Before we go any further, it is helpful to know where to look for help. We can do that with the `man` command.

```
man ls
```

This will give you instructions on what is available to you regarding that specific command. Once you have type `man`, at the bottom of your screen you should see a

```
:
```

This signifies that it is waiting for a command from you. Type `q`.

You will notice that everything is cleared off your screen; this happened because we quit the prompt. Let's open it up again. This time type `/-F`. That performs a search in the commands, similarly to typing `Cmd-F` or `Ctrl-F` when you are searching in a web page.

You can scroll with the arrow keys or by typing `f` or `b` at the `:` prompt.

Now type `h`. This is worthwhile to remember because it shows you all the help commands that you would need for navigating the manual of a given function. The `^` symbol indicates that you should press the `ctrl` key to activate this function. There are many types of optimizations, but this covers the majority of them.

Motivation

We have touched on a few commands such as `ls`,`pwd`, and `man`, and you may be wondering why we would use commands when it is much faster to use a mouse. Although that may be true with what we have covered so far, this is not the case when you are navigating a server in the cloud that does not have a graphical user interface (GUI).

This may be surprising. Not every computer has a GUI because these are quite expensive; the computer must use a lot of resources to have a GUI available, and many servers do not provide this kind of interface. With time, you will likely come to appreciate the command line. It allows you to do what you want in a precise way and makes it extremely easy to automate a set of instructions.

You should also be aware that you will need to use it a fair amount as a data scientist. The vast majority of your work will involve the command line in one way or another.

Basic Commands (cont.)

Let's look at some of the more basic commands. We will navigate to the desktop and create a text file.

```
cd ~/Desktop
```

`cd` stands for *change directory* and allows you to navigate the file system. This should switch us into the desktop, which we can check with

```
pwd
```

This should print that we are on the desktop. Now we can type our next commands.

```
mkdir test_creation
touch my_file.txt
echo "Hello From the Command Line" >> my_file.txt
mv my_file.txt test_creation
echo "This is my second echo command" >> test_creation/my_file.txt
cat test_creation/my_file.txt
```

Notice I do not use spaces very often. Spaces can become cumbersome at the command line and must be escaped with a \\ character. I never use spaces in my file names any more, and with time you likely won't either.

Now let's review what happened line by line. First we created a directory with the `mkdir` command. This is on our desktop, and we should be able to see it if we navigate there with our mouse. Next we created a text file on our desktop with the `touch` command, which creates a simple plain text file that is empty. Next we used the `echo` command to append a string to `my_file.txt` (the one we just created).

`echo` simply repeats what you send it. If you type echo "Hello there", then it will print "Hello there"; however, by using `>>` we append it to `my_file.txt`. This allows us to easily append more information, which we will do shortly.

Now we are doing something that we have not done before. By using the `mv` command, we move the direction from location 1 to location 2. This puts our `my_file.txt` into the folder that we created at the command line.

The next echo command acts similarly to the way it did before but echos the string into the file that we just moved into the folder. We can see the result of that in the next line.

`cat` is a way of reading the contents of a file. When we execute this command, it will print everything in the following file(s) that are passed to it as arguments. It should print

```
$ cat test_creation/my_file.txt

Hello From the Command Line
This is my second echo command
```

We could also change directories into our new `test_creation` directory since we will be doing more in there, and can print the contents from there.

```
$ cd text_creation
$ cat my_file.txt
```

```
Hello From the Command Line
This is my second echo command
```

What is powerful about cat is that it can read more than one file at once, concatenating one onto another. Let's take a look at how that works with the echo command.

```
echo "This is the second file" > second_file.txt
```

Notice that there is only one `>` in that code snippet. When we use only one `>`, we overwrite everything in the file.

```
cat second_file.txt
```

will print out what we just wrote in our echo statement. (Notice how it also created the file for us because it didn't exist.)

Now we can run

```
cat my_file.txt second_file.txt
```

and take it to the next level with

```
echo "Edited first file to be this" > my_file.txt
cat my_file.txt second_file.txt > all_contents.txt
cat all_contents.txt
```

Now let's make a copy of our all_contents.txt file back on our desktop. We would do this as follows. *Same issue with italic as above.*

```
cp contents.txt ../contents.txt
```

Notice that the new file has appeared on your desktop. Open it and check that the contents are the same.

The `cp` command is the copy command where we copy from one place to another. However, you will notice that we copied to a `../` directory. We did this because we are using a *relative* path as opposed to an *absolute* path.

Relative vs. Absolute Paths

These are two important but straightforward distinctions. A relative path is one relative to your current location. The command line knows which directory we are currently in, so when we wrote `cd test_creation` from the desktop, it knew that we were changing into a directory relative to our current one. An alternative that is sometimes required is an absolute path. This is what you will see printed when we run `pwd`. That gives us the entire path. When we were at the desktop, we could have equivalently used the absolute path, which would be something like `cd /users/user_name/Desktop/`; however, writing the whole path is cumbersome–thus, the relative path.

Now that we have covered the distinction between these two paths, let's return to `../`.

Basic Commands (cont.)

Let's go ahead and run in our shell

```
cd ..
ls .
ls test_creation
ls ..
```

This will give us some interesting output. You may have guessed that `..` is short for "up one directory." If we are in a folder contained in another folder, we can use `cd ..` to jump up one directory in the tree (like `test_creation` to Desktop).

`.` refers to the current directory, so `ls` is simply short for `ls .`. As you may have seen in the `ls` help, we can pass a directory to `ls` to list its contents. This allows us to list everything in `test_creation` with `ls test_creation` or in the directory above Desktop with `ls ..`.

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.

```
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 the base commands, but I wanted to show you one last tip. 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 scripting language called Bash.

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`.

```
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.