Unix Shell

```
$ echo "Data Sciences Institute"
$ echo "by: Rachael Lam"
```

Unix

What is Unix?

Unix was created in 1970 and since then has branched into other versions including Linux. Linux was created from Unix with very similar features, although there are some minor differences in commands.

Unix shells - more specifically bash - is a powerful tool for quickly and easily navigating and manipulating files, scaling automated tasks, accessing Git and processing data.

So what is the shell?

The shell is any user interface/program that takes an input from the user, translates it into instructions that the operating system can understand, and conveys the output back to the user.

There are various types of user interfaces:

- graphical user interfaces (GUI)
- touch screen interfaces
- command line interfaces (CLI)

And what is bash?

We'll be focusing on command line interfaces (CLI), more specifically bash, which stands for Bourne Again SHell.

We'll also need a terminal emulator to interact with the shell. This is most likely called terminal on our menu.

Let's get started!

First, we'll open our terminal. As mentioned earlier, this is most likely called terminal and can be found by searching our computer, which on a Mac would be through cmd + space

Let's take a look at the terminal. What do we notice?

- last login
- name
- location
- shell

Looking at the Shell

If we type echo $SHELL in our terminal, the output will tell us what shell we are working with. Most often, our shell will already be bash but in newer Macs, it could be zsh which is almost identical to bash. We can also see where bash is located by typing:

- whereis bash
- whence bash
- which bash

Let's start with a few commands and see what happens in our terminal.

```
$ echo Rachael
$ date
$ cal
$ lksjfs
```

- What happens when we type something that does not exist?
- What happens with errors?

Navigate Files / Directories

Directories

Let's try three commands that help us navigate our system:

1. First, let's run the code below and see what happens:

```
$ pwd
```

`pwd` prints our working directory. If we ever need to know where we are, we can execute this command.

2. Now, let's run the code below and see again what happens:

```
$ cd
```

By default, `cd` changes your working directory to your home directory. You can also use `cd` to set your working directory by including the desired pathname

```
$ cd Desktop
```

In the previous example, we were able to just state `Desktop` because it is a directory in our working directory. If we changed our working directory to `Desktop`, and then wanted to change it again to a directory in `Desktop`, we could again just specify the folder.
If we wanted to change the working directory to a directory outside of our working directory, we would need to specify a pathname:

```
$ cd /Users/rachaellam/Desktop
```

3. To know what files and folders exist in our working directory, we can use the code below:

```
$ ls
```

We can add a pathname at the end to list the contents of a specified directory.

Paths

As we've seen, directory names separated by slashes are paths. There are two types of paths, *absolute* and *relative*.

- An absolute pathname begins at the root directory and includes each directory, separated by slashes until the desired directory or file is reached.
- A relative pathname starts from the working directory and uses symbols `.` or `..` to represent relative positions in the file tree.

Using `cd` and `pwd` let's take a look at how we can use absolute and relative pathnames.

```
$ cd /usr/bin
$ pwd
```

```
$ cd /usr
$ pwd
```

```
$ cd ..
$ pwd
```

Working with Files / Directories

We're going to learn some basic commands to begin some preliminary coding. We'll also be using these throughout the module, so it's important to understand how they work now:

- create directory mkdir
- create file touch
- copy cp
- move and rename mv
- remove rm

Commands

mkdir

First let's make a directory. It's important to remember what directory you're working in currently, because that's where the new directory will be made. Let's assume for now, we're working on our desktop.

```
$ mkdir directory
```

We can also create multiple directories at the same time:

```
$ mkdir dir1 dir2 dir3
```

touch

We can also make new files from the command line. This is particularly useful when we want to make scripts, which we'll learn a bit later. Using touch, we can make a new file in our working directory.

```
$ touch file1
```

We can also create a specific file type by adding the extension:

```
$ touch file1.sh
```

cp

Now we're going to copy a file that we have on our desktop. It can be any file but remember to include the extension or if it has multiple characters, special characters and spaces, to wrap it in quotes.

```
$ cp file1 file2
```

We can also copy files or directories into a directory.

```
$ cp file1 dir1
```

And all files from one directory into another using wildcards:

```
$ cp dirl/* dir2
```

What does the `/*` in this command mean?

There are some useful `—options` that accompany `cp`:

Option	Description
-i	Before overwriting an existing file, prompt the user for confirmation.
-R	Recursively copy directories and their contents.
-v	Display informative messages as the copy is performed.

mv

The mv command enables us to move and rename files and directories, depending on how it's used. In th example below, `mv` renames file1 to file2.

```
$ mv file1 file2 (Renames file1 to file2)
```

Here, `mv` moves file1 to dir1

```
$ mv file1 dir1 (Moves file1 to dir1)
```

We can also move directories into other directories:

```
$ mv dir1 dir2
```

In this case, if `dir2` **exists**, `dir1` will be moved to `dir2`. If `dir2` does not exist, it will be created and `dir1` will be moved to the newly created `dir2`. In both casesm the entire directory will be moved to another/new directory, rather than the contents.

Let's say we're in the directory `Desktop` and we just moved `file1` into `dir1` but now we want to put it back in `Desktop`. How would we move a file out of a directory into another one? Unfortunatly we **can't** just say

```
$ mv file1 Desktop
```

because `file1` does not exist in `Desktop` any more and the command will try and rename `file1` to `Desktop`.

The answer involves using pathnames and the tilde `~` notation:

```
$ mv dir1/file1 ~/Desktop
```

If we just wanted to move `file1` into `dir2` (if `dir2` is in our working directory), we could type:

```
$ mv dir1/file1 dir2
```

What if we want to move just the contents of dir1 to another directory rather than the whole folder? HINT: it is very (exactly) similar to copying (`cp`).

```
$ mv dir1/* dir2
```

This is a combination of the directory `dir1`, pathnames `/` and wildcards `*`. Here, `di1/*` takes the all the contents of `dir1` and puts it in `dir2`.
We could also use the same techqniue to specify certain files to move rather than all of them. How do you think this would be done?

Questions

- We're starting to combine our knowledge of files, directories and pathnames with some basic commands. How do we feel up to this point?

rm

To remove files we use the command `rm`. Because we're now deleting files, it's important that you're sure of what you're deleting because **there is no way to undo**. Fortunately!! there are ways to do this.

```
$ rm file1
```

Without specifying any `-options`, `file1` will be deleted without any feedback.

To ensure we want to delete something, we can use the option `-i` (interactive) that we learned earlier.

```
$ rm -i file1
```

This will prompt a question asking us if we want to delete `file1`. We can respond with `y` (yes) or `n` (no).

If we want to delete a directory, we need to use the option `-r` (recursive) as we did when copying (`cp`). This will recursively delete everything inside of the directory and the directory itself.

```
$ rm -r dir1
```

If we're specifying multiple deletions and a directory does not exist, the shell will tell us. If we don't want that message, we can add the `-option`, `-f` (force). Force will override `-i` if it is included.

1. How do you delete multiple directories?
2. What happens if you delete multiple directories with `-i`?
3. What happens if you delete multiple directories with `-i` but one does not exist?

Remember, it's extremely important to remember that you cannot undo `rm`. This means, if you start using wildcards to specify filenames and don't include `-i`, you could delete things by accident. For example, let's say you want to delete all `.txt` files in a directory:

```
$ rm *.txt
```

If you accidently add a space between `*` and `.txt`,the `rm` command will delete all the files in the directory and then try to find a `.txt` file which does not exist because it delete everything.