Lecture 1:

Course overview + the shell

Topic 1: The Shell

What is the shell?

Computers these days have a variety of interfaces for giving them commands; fancyful graphical user interfaces, voice interfaces, and even AR/VR are everywhere. These are great for 80% of use-cases, but they are often fundamentally restricted in what they allow you to do – you cannot press a button that isn't there or give a voice command that hasn't been programmed. To take full advantage of the tools your computer provides, we have to go old-school and drop down to a textual interface: The Shell.

All platforms you can get your hand on has a shell in one form or another, and many of them have several shells for you to choose from. At the core all the terminals are roughly the same: they allow you to run programs, give them input, and inspect their output in a semi-structured way.

This lecture is focussed on the Bourne Again Shell, or "bash".

Using the shell

```
missing:~$
```

This is the main textual interface to the shell. It tells you that you are on the machine missing and that your "current working directory", or where you currently are, is ~ (short for "home"). The $ tells you that you are not the root user. At this prompt you can type a *command,* which will than be interpreted by the shell. The most basic command is to execute a program:

```
missing:~$ date
Fri 10 Jan 2020 11:49:31 AM EST
missing:~$
```

we executed the date program, which (perhaps unsurprisingly) prints the current date and time. The shell then asks us for another command to execute.
we can also execute a command with arguments:

```
missing:~$ echo hello
hello
```

In this case, we told the shell to execute the program echo with the argument hello. The echo program simply prints out its arguments. The shell parses the command by splitting it by whitespace, and then runs the program indicated by the first word, supplying each subsequent word as an argument that the program can access. If you want to provide an argument that contains spaces or other special characters (e.g., a directory named "My Photos"), you can either quote the argument with ' or " ("My Photos"), or escape just the relevant characters with \ (My\ Photos).

The shell is a programming environment, just like Python or Ruby, and so it has variables, conditionals, loops, and functions. When you run commands in your shell, you are really writing a small bit of code that your shell interprets. If the shell is asked to execute a command that doesn't match one of its programming keywords, it consults an environment variable called $PATH that lists which directories the shell should search for programs when it is given a command:

```
missing:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
missing:~$ which echo
/bin/echo
missing:~$ /bin/echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

When we run the echo command, the shell sees that it should execute the program echo, and then searches through the : -separated list of directories in $PATH for a file by that name. when it finds it, it runs it (assuming the file is *executable*; more on that later). We can find out which file is executed for a given program name using the which program. We can also bypass $PATH entirely by giving the path to the file we want to execute.

Navigating in the shell

A path on the shell is a delimited list of directories; separated by / on Linux and macOS and \ on Windows. On Linux and macOS, the path / is the "root" of the file system, under which all directories and files lie, whereas on Windows there is root for each disk partition (e.g., C:\).

Assuming you are on a Linux filesystem for this.

A path that starts with / is called an *absolute* path. Any other path is a relative path. Relative paths are relative to the current working directory, which we can see with the pwd command and change with the cd command. In a path, . refers to the current directory, and .. to its parent directory:

```
missing:~$ pwd
/home/missing
missing:~$ cd /home
missing:/home$ pwd
/home
missing:/home$ cd ..
missing:/$ pwd
/
missing:/$ cd ./home
missing:/home$ pwd
/home
missing:/home$ cd missing
missing:~$ pwd
/home/missing
missing:~$ ../../bin/echo hello
hello
```

Notice that our shell prompt kept us informed about what our current working directory was. You can configure your prompt to show you all sorts of useful information.

In general, when we run a program, it will operate in the current directory unless we tell it otherwise. For example, it will usually search for files there, and create new files there if it needs to.

To see what lives in a given directory, we use the ls command:

```
missing:~$ ls
missing:~$ cd ..
missing:/home$ ls
missing
missing:/home$ cd ..
missing:/$ ls
bin
boot
dev
etc
home
...
```

Unless a directory is given as its first argument, ls will print the contents of the current directory. Most commands accepts flags and options (flags with values) that start with – to modify their behaviour. Usually, running a program with the -h or –help flag (/? on Windows) will print some help text that tells you what flags and options are available. For example, ls –help tells us:

```
-l                                use a long listing format
```

```
missing:~$ ls -l /home
drwxr-xr-x 1 missing  users  4096 Jun 15  2019 missing
```

This gives us a bunch more information about each file or directory present. First, the d at the beginning of the line tells us that missing is a directory. Then follow three groups of three characters (rwx). These indicate what permissions the owner of the file (missing), the owning group (users), and everyone else respectively have on the relevant item. A – indicates that the given principal does not have the given permission. Above, only the owner is allowed to modify (w) the missing directory (i.e., add/remove files in it). To enter a directory, a user must have a "search" (represented by "execute": x) permissions on that directory (and its parents). To list its contents, a user must have read (r) permissions on that directory. For files, the permissions are as you would expect. Notice that nearly all the files in /bin have the x permission set for the last group, "everyone else", so that anyone can execute those programs.

Some other handy programs to know about are mv (to rename/move a file), cp (to copy a file), an mkdir (to make a new directory).

For more information about a program's arguments, inputs, outputs, or how it works in general, give the man program a try. It takes as an argument the name of a program, and shows you its *manual* page. Press q to exit.

```
missing:~$ man ls
```

Connecting programs

In the shell, programs have two primary “streams” associated with them: their input stream and their output stream. When the program tries to read input, it reads from the input stream, and when it prints something, it prints to its output stream. Normally, a program’s input and output are both your terminal. However, we can also rewire those streams!

The simplest form of redirection is < file and > file. These let you rewire the input and output streams of a program to a file respectively:

```
missing:~$ echo hello > hello.txt
missing:~$ cat hello.txt
hello
missing:~$ cat < hello.txt
hello
missing:~$ cat < hello.txt > hello2.txt
missing:~$ cat hello2.txt
hello
```

You can also use >> to append to a file. Where this kind of input/output redirection really shines is in the use of *pipes*. The | operator lets you “chain” programs such that the output of one is the input of another:

```
missing:~$ ls -l / | tail -n1
drwxr-xr-x 1 root  root  4096 Jun 20  2019 var
missing:~$ curl --head --silent google.com | grep --ignore-case content-
length | cut --delimiter=' ' -f2
219
```

More details on advantage of pipes further.

A versatile and powerful tool

On most Unix-like systems, one user is special: the “root” user. The root user is above (almost) all access restrictions, and can create, read, update, and delete any file in the system.

Using the sudo command. As its name implies, it lets you “do” something “as su” (short for “super user”, or “root”). When you get permission denied errors, it is usually because you need to do something as root. Though make sure you first double-check that you really wanted to do it that way!

One thing you need to be root in order to do is writing to the sysfs file system mounted under /sys. Sysfs exposes a number of kernel parameters as files, so that you can easily reconfigure the kernel on the fly without specialized tools. **Note that sysfs does not exist on Window or macOS.**

For example, the brightness of your laptop’s screen is exposed through a file called brightness under

```
/sys/class/backlight
```

By writing a value into that file, we can change the screen brightness. Your first instinct might be to do something like:

```
$ sudo find -L /sys/class/backlight -maxdepth 2 -name '*brightness*'
/sys/class/backlight/thinkpad_screen/brightness
$ cd /sys/class/backlight/thinkpad_screen
$ sudo echo 3 > brightness
An error occurred while redirecting file 'brightness'
open: Permission denied
```

This error may come as a surprise. After all, we ran the command with sudo! This is an important thing to know about the shell. Operations like |, >, and < are done by the shell, not by the individual program. echo and friends do not "know" about |. They just read from their input and write to their output. In the case above, the *shell (*which is authenticated just as your user) tries to open the brightness file for writing, before setting that as sudo echo's output, but is prevented from doing so since the shell does not run as root. Using this knowledge, we can work around this:

```
$ echo 3 | sudo tee brightness
```

Since the tee program is the one to open the /sys file for writing, and it is running as root, the permissions all work out. You can control all sorts of fun and useful things through /sys, such as the state of various system LEDs (your path might be different):

```
$ echo 1 | sudo tee /sys/class/leds/input6::scrolllock/brightness
```

NOTE: sudo su will transition you to "root" terminal with # indicating the same.