UNIX TEACHING OUTLINE

Place: CDT space 402
Time: 10 am
Helpers: Katerina and Jazz

Remember: split screen with big text / encourage stickies / explain-type-repeat / write commands up / do sketches / don't use the word 'just' / normalise errors / work in pairs

INTRODUCTIONS() (15 minutes)

- Arrange so sitting in pairs for pair programming
- Show PPT (motivation, outline, key definitions)
    - Hi, I'm Lucy and I'm a PhD student here at Imperial and I model – using computers – the behavoiur of electrons in materials. I first started using Unix about ten years ago, as an undergraduate, and during my PhD I've used it every day. I find it extremely useful for my work – we'll come back to that later. First I'd like to introduce Katerina and Mack who are the helpers for this lessson. They will be floating around, looking over your shoulder, they are not trying to read your personal emails, they just want to check I've got the pace right, that I'm not going to fast and leaving people behind. Please help them by using the stickies.
    - So why should we learn the Unix shell? Well it can be used to automate repetative tasks
      - Many of us in the course of our research will be asked to do the same piece of analysis on similar data sets. Manually repeating the same task many times is in the most part, not an efficient use of time, it can also lead to silly mistakes. For example, you may be a marine biologist who has collected data on 800 different samples of seaweed. You need to run this data through an analysis programme your superivosr has written. This prgramme takes ten minutes to run. So you start the programme, wait ten minutes, start it again on the next ten minutes, wait ten minutes…it's going to take a long time. Using the Unix shell you could write something called a loop, which we will cover today, and the process will be repeated for each sample without you doing each one manually.
      - Make your science more easily reproducible. We will look at something called shell scripting, this is where you save the commands you have used in a file for later use. So if someone asks, how did you manipulate the data in those files to get to that result, you can hand over your scripts, and the steps you took are contained in there.
      - Supercomputers are very large, powerful computers which are most often, away from the site you are physcially working. Imperial has supercomputers in ??, the UK national supercomputer Archer is based in Edinburgh, I've even used a supercomputer called SiSu in Finland. You communicate with these supercomputers using Unix shell commands. So if you think you might want to use HPC in the future, then you will need to know the basics of Unix shell.
    - Finally, and this is what got my into computing in the first place, you can use the unix shell to feel like a Hacker. Who has seen this film - Hackers? Brilliant film, made in 1995,featuring angelina and tommy lee millller as teenage hackers, using, of course the unix shell for their hacking. So if, like me, you think this kind of thing is cool, then you should learn the unix shell.
    - 14 commands this morning– it's a manageable amount, Building computer skills takes time, we cannot teach you the whole of the Unix shell in a half day. We are not trying to teach you everything but I do want to achieve two things: I would like you to walk away with at least one command or group of commands that would be useful for speeding up your workflow, and I'd like you to have the Confidence to learn more about this in your own time.
- First I'd like to outline some key vocab. Unix is a type of operating system – like the Windows operating system that you might use. If you are using Mac, you are using a Unix operating system.
  - The Unix shell, which we are learning about today, is a command line interface to the Unix system. Command line because there is no graphical user interface like when you run word , to use this interface we type commands. By typing commands we get it to run built in programmes.
  - A number of different unix shells have been developed, the one we are using, which is the most popular is called the bash shell. I tend to use the bash and unix interchangeably for this reason.
  - A terminal is the programme you open on your computer which runs the shell.
  - The prompt – known as command prompt – is this ending in the dollar sign. It marks the beginning of the command line.

MAIN() (2 hours 10min + 30min break)

- **Part one: Introducing the shell** (10 minutes)
  - (open up Terminal) So I click on the terminal icon and this gives me access the Unix shell. Can everyone open up their shell? Stickie when you are done.
  - We are going to use Live coding. So I type in my screen, and you follow

```
$ ls –F /
```

  - 3 part: command, flag and arguments. (Draw on board the 3 parts) The command corresponds to a short programme, in this case it is the ls programme. The flag adjusts the behaviour of the programme and the argument tells the command what to operate on.
  - When you are typing terminal commands you must copy things exactly – spaces are important, capitalization is important. Lower case is not equivalent to upper case.
  - After each terminal command you press return and the shell will evaluate it, print the results to the screen, then when it is ready for the next command it will give you the prompt again. That tells you it is waiting for the next input.
  - Ls is the list programme which lists the contents of a directory (directory = folder). The –F flag tells it to give me a bit more information about directory contents. So it puts a slash after each entry which is a directory. If I do it without, it doesn't give me the slashes

```
$ ls /
```

  - The argument is a forward slash. This is used to denote the root directory of my computer. (Draw on the board the root and it's sub directories) The root directory holds everything else. We refer to it using a slash. Bin=binaries, where programmes are stored, tmp is short for temporary it is for temporary files the computer does not need to store long term. Users contains the personal directories for each user, and as users this is where we spend most of our time.
  - What happens if we type something incorrectly?

```
$ ls–F /
```

  - (Write ls onto the commands list)

- **Part two: Navigating directories** (15 minutes +10 minutes tasks)
  - Let's find out where we are sitting in the directory structure. We do this using the command pwd.

```
$ pwd
```

  - The computer tells us we are sitting in /Users/learner
  - (Draw diagram) So we are sitting here in the learners directory which is itself in the Users directory
  - For every command there is a built-in help page you can access using the command

```
$ man pwd
```

  - We can move up and down this page using the up and down keys. Q to exit.
  - If we use ls without an argument it will list the contents of our current directory

```
$ ls
```

  - What command could we use to list the contents of the Desktop directory? Ls Desktop

- Now the third command for this section: cd. It is short for change directory. So if I want to move into my desktop I would use

```
$ cd Desktop
$ pwd          # I'm in the right place
```

- We need to download some files for the workshop today. I'm going to show you how to do this using the command curl. Is everyone sitting in their desktop? Good.

```
$ curl -O https://swcarpentry.github.io/shell-novice/data/data-shell.zip
```

- This is the longest thing you will need to type today!
- Go to your Finder window and you should see the zip file. Unzip the file (in Mac I think you can double click) then go back to the terminal. Now you should see the folder in there (Add to drawing)

```
$ ls -F
```

- I can see what is in this folder (Add to drawing)

```
$ ls data-shell
```

- And what is contained within data

```
$ ls data-shell/data
```

- And if I wanted to move into this directory I could use

```
$ cd data-shell/data
```

- We are now sitting in data. How about if we want to go up, into data-shell?

```
$ cd data-shell    # doesn't work
$ cd ..
```

- This command always moves us up one directory – to the parent of the current
- If we run ls we don't see it as an option. That is because it is a hidden file. To see hidden files we need to run ls with the –a short for all flag

```
$ ls -a
$ ls -F -a        # I can combine flags
$ ls -Fa          # Or we can combine like this
```

- If we type cd without an argument what happens?

```
$ cd
$ pwd            # we can find out with pwd – it takes us to our home directory
$ cd /Users/learner/Desktop/data-shell     #  TAB COMPLETION to go back to our project
```

- Fab, we have covered the three commands cd, pwd and ls
- Now I'd like you to go to the website https://swcarpentry.github.io/shell-novice . This contains all the resources for today. Go to episode x , read this section and do the three exercises in paris: you have ten minutes.
- We have covered how to navigate through our computer using cd, pwd and ls.
- What questions do you have on this?
- Now lets look at creating and moving files and directories.

- **Part three: Creating and editing files and directories** (15 minutes + 10 minutes tasks)
  - Let's check that we are in the correct place

```
$ pwd
$ ls    # and see what it contains
```

  - I want to create a directory called thesis. I use the command

```
$ mkdir thesis
```

  - (Draw on diagram) This is a relative path because it does not have a leading slash. So it is made relative to our current position, it's made in the directory we are currently sitting in.
  - Note that when you're working with unix, it is much easier if file and directory names do not have any whitespace. Stick with letters, dash and underscore. (on board)(new command)
  - Let's move into the thesis folder and create a file

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

  - Nano is a command which opens up the nano text editor, there are plenty of other editors but we use nano as it is one of the most accessible ones
  - As you can see, it looks nothing like word! It really is just a text editor.

```
Trust your technolust
Hack the planet                    # These are quotes from Hackers
```

  - Ctrl-o to save, return to accept, ctrl X to leave
  - Is everyone happy?
  - My thesis probably shouldn't be based around quotes from Hackers, so I'll delete this draft using the comman

```
$ ls
$ rm draft.txt
$ ls             # And it's gone. WARNING this is forever (no rubbish bin)
```

  - Now I'll move back into the data-shell directory and try to remove the thesis directory

```
$ cd ..
$ rm thesis
$ rm –r thesis  # need to use –r for recurring when deleting a directory
$ mkdir thesis   # let's make it again
$ rm –r –I thesis  # this is a safer way of doing it – interactively.
```

  - Let's create the file and directory one more time.

```
$ pwd
$ mkdir thesis
$ nano thesis/draft.txt   # note we're running nano with path thesis/draft rather than cd thesis
"With great power comes great responsibility"
$ ls thesis
```

  - Say I want to change the name from draft.txt to quotes.txt then I use (write on board)

```
$ mv thesis/draft.txt thesis/quotes.txt
$ ls thesis
$ mv thesis/quotes.txt .     # . is a special name for the directory we are currently in
$ ls thesis
$ ls
```

- Again, you need to careful as mv is DANGEROUS. If you had another file called quotes.txt, it would overwrite this file.
- If I want to copy my file quotes.txt I use (write on board)

```
$ cp quotes.txt thesis/quotations.txt
$ ls
$ ls thesis
```

- We have now went through the commands for creating / moving and copying files. Go to the website and read this, then discuss with your partners the questions below. Do as many as you can in 10 minutes.

- **Part four: combining commands** (15 minutes + 10 minutes activities)
  - What questions do you have from the previous section?
  - Ok, this is the final section before coffee break and we are getting on to the good stuff. This is where unix gets powerful, because it lets you chain together lots of simple commands to make a powerful programme.

```
$ pwd     # you should be in data-shell
$ cd molecules
$ ls
$ wc *.pdb
```

- Wc is word count and the asterix is a wildcard. It means match all files that end in .pdb

```
$ wc -l *.pdb
```

- This now shows the number of lines per file. Which file is the shortest? Easy when we have six files, but what if we had 6000 files?
- First step would be run this command

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

- Normally this command would output to our screen like we saw before. But this time it redirects the output to the file lengths.txt DANGER could overwrite.
- We can see what lengths.txt contains using the command cat which is short for concatenate. It prints the contents of one file to another. Here I only have one filename so it prints the fie to the screen.

```
$ cat lengths.txt
```

- Now we can sort the contents of this file using the command sort (write it up)

```
$ sort -n lengths.txt
```

- -n to sort numerically. This does not change the contents of the file.
- We can pipe the output into another file sorted-lengths.txt

```
$ sort –n lengths.txt > sorted-lengths.txt
```

- And if we want to look at the top line of this file

```
$ head –n 1 sorted-lengths.txt
```

- (write it up) Or we can look at the top 3 lines using

```
$ head –n 3 sorted-lengths.txt    # TAB COMPLETION
```

- Now we could do this process for 20,000 files to find the shortest one quickly. We have created this file "sorted-lengths.txt" which we don't really need. We can remove it from our workflow by using a pipe

```
$ sort –n lengths.txt | head –n 1
```

- The output of the command on the length is the input of the command on the righted
- What about getting rid of this intermediate file lengths.txt? To create that we used

```
$ wc –l *.pdb | sort –n
```

- Instead of redirecting this to a file we can pipe it to sort –n.
- We could pipe this output to another command head. So the full workflow for finding the longest file in a directory would be

```
$ wc –l *.pdb | sort –n | head –n 1
```

- If you are finding it hard to understand what is happening here, it might be easier to think of it using mathematical notation:
  Similar to log(3X) where you would multiply x by 3 then take the log
  Head –n 1(sort –n(wc –l *.pdb))) : we are doing the inner bracket first then this is input for the next function and so on…
- Now I've only shown you a handful of commands but can you imagine how powerful it is as you learn more and more commands and combine them in this way?
- This way of approaching programming is called modular. Where a complex programme is built from lots of smaller, simple units. It is the key to the success of Unix, that it was built in this way. This type of modular programming can be very effective and it is something I always aim for when I am writing a programme in other languages like Python for example.
- Ok those are all of the commands. I would like to finish this section with you and your partner to discuss the answer to "Pipe reading comprehension" . Write what you think final.txt will contain . You have three minutes.
- Discuss . What questions? then coffee

- **Part five: Repeating commands with loops** (15 minutes + 10 minutes)
  - We are now going to learn about loops. Loops allow us to repeat the same command many times over. For this section we are going to work in the creatures directory

```
$  pwd
$  cd     # remember if you are lost you can cd to you home directory.
$ cd Desktop/data-shell     # then from there you can find the right place
$ cd creatures
$ ls     # there are two creatures in there
```

  - We want to modif y these files and keep a version of the original. For two files, we could do it by hand. But what if we had 50? It starts taking a long time…we could try

```
$ cp .dat original-*.data
```

- But that doesn't work because it expands as (write on board) and when cp has more than 2 arguments it expects the last to be a directory it can copy all the files to.
- Instead we can use a loop (on the board). Here is an example loop.

```
$ for filename in basilisk.dat unicorn.dat
> do
>     head -n 3 $filename
> done
```

- Indentation not necessary but good for readability
- For tells tells the shell to repeat the comman between do and done
- In this case it will be repeated twice: once for basilisk.dat and once for unicorn. Dat
- The first time it will run ….the second time it will run …..
- Filename is what I call a dummy variable. It doesn't matter what its called as long as it is the same here and here. I could replace it with x (demonstrate).
- The dollar sign is important as it tells the shell that filename is a variable and that it needs to substitute a value in it's place
- So this is all exactly equivalent to (write it up) but for a 1000 files would be infinitely quicker.
- Lets type it in

```
$ Type the above in
```

- See the more than sign. The shell is telling us it is expecting more input before it evaluates the command.
- The output is as expected
- Write on board We don't need to write this as separate lines , we could write it as

```
$ for filename in basilisk.dat unicorn.dat ; do head -n 3 $filename; done
```

- They are exactly equivalent. Multi-line commands can be separated with a semicolon instead

```
$ Type the above in
```

- Challenge: who can tell me what the following loop can do? Feel free to discuss with your partner. I want you to work it out without running the code – you can use the built in help though.

```
$ for filename in *.dat
$ do
$    echo $filename
$    head -n 100 $filename | tail -n 20
$ done
(it will give lines 81 to 100 of each file)
```

- We haven't discussed the command tail yet but remember you can use

```
$ man tail
$ man echo
```

- Now a challenge for you and your partner. Remember the original task – write a loop which copies each filename.dat orig-filename.dat . check you have done it with the ls command. Once you have finished then put green stickie so I know we can move on.
- We have finished the loops section. What questions?

- **Part six: Saving commands for later** (15 minutes + 10 minutes)
  - Now the final section. We are going to look at how to save a group of commands into a shell script - which is a small programme - so they can be run again later using a single command, or shared with other people for them to use. I mentioned at the start reproducibility, having your analysis contained within a shell script is one way to ensure reproducible research.

```
$ pwd   $ we want to be in the molecules directory
$ cd ../molecules $ go back into data shell then into molecules directory
$ nano middle.sh
```

  - we have created the file middle.sh the sh prefix tells computer it is a bash shell script.

```
Head –n 15 octane.pdb | tail –n 5
```

  - This selects lines 11-15 of the file octane.pdb

```
$ ls
$ bash middle.sh
```

  - Output is the same as if we ran the command directly.

```
$ nano middle.sh
head –n 15 $1 | tail –n 5
```

  - $1 is a special variable which stands for the first argument on the command line.

```
$ bash middle.sh octane.pdb
```

  - The same result. Can do a different file now

```
$ bash middle.sh pentane.pdb
```

  - Lines 11 to 15 of pentane file. What about if we want to decide the part of the file we want to print out?

```
$ nano middle.sh
head –n $2 $1 | tail –n $3   # first arg filename and arguments 2 and 3 tell me the lines
$ nano middle.sh pentane.pdb 20 5   # will extract lines 16 to 20 of pentane
```

  - If somebody else uses this programme it might not be obvious what it does so we should add comments

```
$ nano sorted.sh
# select lines from a middle of a file
# Usage: bash middle.sh filename end_line num_lines
```

  - Here we are processing one file at a time, but what if we wanted to process many? For example, to sort our .pdb files by length we would use

```
$ wc –l *.pdb | sort –n
```

  - But it would only ever sort .pdb in current directory. A more flexible script would be

```
$ nano sorted.sh
# Sort filenames by their length
```

```
# Usage: bash sorted.sh one_or_more_filenames
wc -l $@ | sort -n


$ bash sorted.sh .pdb ../creatures/.dat
```

- What questions do you have?
- Put up “find the Longest File with a given extension”
- Something that’s not discussed in this workshop but that I find useful for my work is cron jobs. These are shell scripts you set to execute at a particular time at a particular frequency. For instance, I made a shell script that pulls data from the supercomputers I use onto my local computer. The data files are large and can take a while to transfer, but I like having them ready for the start of the working day. Using cron allows me to set this script to run automatically every night at midnight so the files are waiting for me in the morning.

END PPT() (10 minutes)