2025-11-04-strathclyde Shell lesson instructor notes

Summary and Setup

[SLIDE HERE: Links/QR codes]

- Lesson site: https://swcarpentry.github.io/shell-novice/

Introduction to the shell

- The Unix shell has been around longer than most users, including me, have been alive
- It has survived and adapted, *unlike flares, glam rock, Madchester, and the comedy reputation of Friends*, because it is a **very powerful tool for controlling a computer**.
- With the shell you can carry out extremely powerful tasks, often with only a few keystrokes or lines of code.
- You can use it to **automate repetitive tasks** and **combine smaller tasks** into larger, even more powerful **workflows**.
- Importantly for us as scientists, it allows us to easily record what we did (**reproducibility**) and share the methodology (**repeatability**)
- If you want to use high-performance computing (such as ARCHIE-West, here at Strathclyde) in your work - and this is becoming much more necessary as the volumes of biological data increase - you will need to use the shell.
- We'll be working through the filesystem and the shell.
- If you have ever saved files on a computer and know the difference between a "file" and a "directory"/"folder", then you are ready for this lesson.

Setup

- Please ensure that you have access to the Unix shell on your machine
    - University machine: please use Git bash (access via the Start menu)
    - Download/installation instructions: https://carpentries.github.io/workshop-template/install_instructions/#shell
- You will need to download some files to follow this lesson
- **Please see the EtherPad page for a link**
    - https://pad.carpentries.org/2025-10-04-strathclyde
    - Download this file to your `Desktop`
    - Unzip/uncompress the file
        - **NOTE: do not assume that your filesystem has unzipped the file; Windows will navigate into the compressed file using the explorer, without unzipping it!**

[PAUSE: ensure that all learners have access to a bash shell]

- Open a new instance of the shell on your computer

[PAUSE: ensure that all learners can start a bash shell]

Introducing the shell

- There are many ways to interact with a computer. You'll probably be used to methods like:
    - using a **touchscreen**
    - **keyboard and mouse**
    - maybe even voice interaction/**speech recognition**
- The most common method, still, is via a **GUI** (Graphical User Interface), clicking with a mouse through a series of menu interactions
- The GUI approach is **easy to learn**, and it **may be intuitive**
- Unfortunately, **it does not scale well**. What do we mean by that?
    - Suppose you're carrying out a literature search.
    - Imagine you need to copy the affiliations for every author in 1000 papers to a single file, for analysis.
    - You could open each file, copy the affiliations, and paste them into an ever-growing file.
    - But this would take a long time.
    - It would also be prone to error - it's easy to **miss a character** when copy/pasting. You might even **miss an entire affiliation**, or **accidentally skip one or more files**. And **how would you know** if you'd done that? That would be a lot of finicky manual checking, and you'd possibly even **overlook some mistakes**.
- This is where the CLI (Command-Line Interface) approach is especially powerful.
    - These **repetitive tasks can be automated** for a single file, **and applied automatically across many files** - an arbitrary number - extremely quickly.
- You could spend hours manually copy/pasting and checking, or you could spend a minute writing a shell script, and get the job done in seconds.

The Shell

[OPEN THE SHELL ON-SCREEN]

- So **what is the shell**?
- The shell is **a program that runs on your computer**.
- Users, like you and me, can **type commands into the shell**, and the computer will run them.
- We can **use the shell to start other programs**, like modelling software for metabolism or industrial processes, protein structure prediction, or sequence analysis tools.
- The most common shell is called bash (Bourne-Again SHell) and is the default on many systems.

    - *On a Mac, you may see the Z Shell (`zsh`) instead. This works just like bash.*

- The big difference between a GUI and the shell is that **a GUI usually shows you what options/commands are available**.
    - **The shell does not**.
- It can take some time and effort to learn how to use the shell well.
    - **But you can get a long way with only a few commands - the ones we'll learn today.**
- In the shell, you can easily:
    - **combine existing tools and programs into pipelines and workflows**, for automation
    - **combine sequences of commands into *scripts***, for reproducilty
    - **interact with powerful remote computing resources**, like ARCHIE-West (this is an increasingly important skill in modern biology)

Let's get started

- When you open the shell, you will see the **prompt**, which tells you the shell is waiting for your input:

```
(base) lpritc@Rodan-2 shell %
```

- **Your shell will look different to mine**.
    - The CLI prompt is customisable, and I have some extra information in my shell because it's useful for me in my day-to-day work.
    - The variation does not matter for the purpose of this lesson.
    - **The important part for this lesson is the `%` or `$` symbol**
    - You should also see a **cursor**, just after the prompt.
        - This might be a flashing block, a static block, an underscore, or something else. The visual appearance can vary depending on which system you are using.
    - If you can see this symbol, everything is fine.
- The shell is waiting for you to type a command.
- Once you have typed a command at the prompt, you execute it using the `enter` or `return` key.
- **Let's try our first command**L `ls`, short for "listing", which lists the current directory contents.
    - Type `ls` and press the `enter`/`return` key

```
(base) lpritc@Rodan-2 shell % ls
instructor_examples/ instructor_notes.md
(base) lpritc@Rodan-2 shell %
```

- Notice that once the command completes, you are presented with a new shell prompt, ready for the next command.

Nelle's Pipeline

[SLIDE HERE: Nelle's Pipeline outline]

- For the lesson we're going to pretend to be a marine biologist called Nelle
- Nelle, i.e. you, has/have just returned from a six-month long survey of the North Pacific Gyre, where you were sampling marine life in the Great Pacific Garbage Patch
    - You have **1520 samples** that you ran through a mass spec to gather the relative abundances of 300 different proteins
    - You need to **run these samples through an analysis tool** called `goostats.sh`
    - You also need to **write up your results by a deadline** at the end of the month.
- If you were to carry out this analysis by hand i**n a GUI, you'd have to select and open a file 1520 times**
    - If `goostats.sh` takes **30s to analyse a file, that's 12 hours of your time** just pointing and clicking.
    - **With the shell you can get the computer to handle all the boring stuff** while you crack on with the paper introduction.
- What you're going to do is:
    - **use the shell to run `goostats.sh`**
    - **use loops** to automate running `goostats.sh` on many different files
- As a bonus, **once this pipeline is up and running, you can use it again on new data**, whenever you collect it - saving you even more time.

[SLIDE HERE: Nelle's Pipeline details]

- Specifically, you're going to:
    - navigate to a file/directory
    - create a file/directory
    - check the length of a file
    - chain commands together
    - retrieve a set of files
    - iterate over files
    - run a shell script containing your pipeline

Navigating files and directories: `ls`

- **[SLIDE HERE: Title Slide]**
- Let's start by seeing how to
    - **move around** on your computer

    - **see what files and directories** you have
    - **specify the location of a file or directory** on your computer

- **[SLIDES HERE: Filesystem Structure]**

- The part of the operating system responsible for managing files and directories is called **the file system**. It organizes our data into:

    - **files, which hold information**
    - **directories (also called 'folders'), which hold files or other directories.**

- Note that **your filesystem will not look exactly like that in the examples**, or like mine.

    - **You will sometimes need to modify the commands on screen so that they work on *your* computer**.

- Let's look at Nelle's home directory (on the slide)

    - The filesystem on Nelle's machine is that shown on screen
    - Notice that it's a hierarchy that **looks like a family tree, or phylogenetic tree** - with the root at the top
    - **The very top directory is the *root directory*** and has the symbol `/` - this directory holds all of the other directories.
    - This directory is the leading slash in `/Users/lpritc`

- In Nelle's *root directory* there are `bin`, `data`, `Users` and `tmp` directories.

    - We know that Nelle's working directory `/Users/nelle` is stored inside the `/Users` directory (this is the first part of the name returned by `pwd`)

- There are several commands for creating, inspecting, and deleting files and directories.

- Let's find out where we are - ourselves - by running a command called pwd (which stands for 'print working directory').

    - Directories are like *places*
    - At any time while we are using the shell, we can be in exactly one place at any time (though we can move from one directory/place to another)
    - The place we are in is called our current working directory.
    - Commands mostly read and write files in the current working directory, i.e. 'here', so knowing where you are before running a command is important.

- **[NEXT SLIDE: User Directories]**

- In the `/Users` directory, we find one directory for each user account on that machine (just like I have `lpritc` as my home directory)

    - Nelle, Imhotep, and Larry
    - Nelle's home directory is `/Users/nelle`

- **Generally, when you open a new shell, you start in your home directory**

- We can see what's in our home directory by using the `ls` command

[OPEN SHELL IN HOME DIRECTORY]

- The `pwd` (print working directory) command in the shell shows you where you are

```
(base) lpritc@Rodan-2 lpritc % pwd
/Users/lpritc
(base) lpritc@Rodan-2 lpritc %
```

- Here the response is `/Users/lpritc` - which is my **home directory** on this machine
    - On Linux, your home directory may look like `/home/nelle`, and on Windows, it might be similar to `C:\Documents and Settings\nelle` or `C:\Users\nelle` - this is normal and reflects the differences between operating systems
- Let's see what's in the home directory:

```
(base) lpritc@Rodan-2 lpritc % ls
Applications/
Applications (Parallels)/
bin/
Calibre Library/
Desktop/
Documents/
Downloads/
Dropbox (Dropbox-Work)/
Google Drive@
iCloud Drive (Archive)/
iCloud Drive (Archive) - 1/
Library/
Movies/
Music/
nltk_data/
opt/
Parallels/
Pictures/
Public/
seaborn-data/
Zotero/
```

- **Your results will be different - this is normal**.
- My shell has colours that differentiate between directories and files - your shell might not do this by default.
    - If you'd like to see the colours, use `ls -F`:

```
% ls -F
```

Command-line options

- You'll see here that we have provided an **option** to `ls`, which tells it in more detail how it should present the contents of the directory
  - *Options* change the behaviour of commands.
- Many commands have additional **options** and **arguments** that you can use to control their behaviour, and there are two standard ways to find out what these are, in the terminal
  - add the `--help` or `-h` option
  - use the `man` manual system

```
# ls --help doesn't work on macOS, but should on Linux and Git Bash
% ls --help
ls: unrecognized option `--help'
usage: ls [-@ABCFGHILOPRSTUWXabcdefghiklmnopqrstuvwxy1%,] [--color=when]
[-D format] [file ...]
```

```
# man ls does work on macOS
% man ls
```

- You'll see that there are long and short options that are equivalent, like `-h` and `--help` - **it makes no difference to functionality which you use**
  - But, if you're **writing a script it can be more legible/reproducible to use the long form** of an option.
- **[SLIDE HERE: `ls` challenge (reverse chronological)]**

Exploring other directories

- We can use `ls` to list the contents of the current directory, and **we can also use it to list the contents of a different directory**
- Let's take a look at the contents of the desktop using `ls -F Desktop` - we tell `ls` the *path* to the location we want to list contents for
  - This should show all the files and directories on your desktop (assuming you are in your home directory)
  - In amongst these should be the `shell-lesson-data` directory that you downloaded
- **[SHOW SHELL]**

```
% ls -F Desktop
[...]
shell-lesson-data/
[...]
```

- We can use the same strategy to look *inside* the `shell-lesson-data` directory, with `ls -F Desktop/shell-lesson-data`
    - We see the two directories: `exercise-data` and `north-pacific-gyre`

```
% ls -F Desktop/shell-lesson-data
exercise-data/      north-pacific-gyre/
```

- We are currently located in our home directories, but we want to be in the `exercise-data` directory, so we can progress with the lesson.
- To do this we use the `cd` (change directory) command, telling it which directory we want to move to
    - We can do this three times, stepping first into `Desktop`, then `shell-lesson-data`, and then to `exercise-data` confirm the move with `pwd`:

```
% cd Desktop
% cd shell-lesson-data
% cd exercise-data
% pwd
/Users/lpritc/Desktop/shell-lesson-data/exercise-data
```

- Now we can see the contents of the folder with `ls -F`

```
% ls -F
[1:33:18]
alkanes/       animal-counts/ creatures/     numbers.txt    writing/
```

- We've moved *down* in the directory tree, but **how do we move up?**
    - We can't use `cd shell-lesson-data` because we can only "see" the current contents of the directory
    - But there is **a special shortcut to move up one directory level**: `..`

```
% cd ..
% pwd
/Users/lpritc/Desktop/shell-lesson-data
```

Other shortcuts

- We can always return to our home directory using the shortcut `~`, which stands for "home directory"

```
% cd ~
% pwd
/Users/lpritc
```

- You can `cd` directly to a location by using its "absolute path" (starting with `/` at the root)

```
% cd /Users/lpritc/Desktop/shell-lesson-data
% pwd
/Users/lpritc/Desktop/shell-lesson-data
```

- And you can use `~` to indicate your home directory as a starting point

```
% cd ~/Desktop/shell-lesson-data/exercise-data
% pwd
/Users/lpritc/Desktop/shell-lesson-data/exercise-data
```

- If you use `ls -a` - the option to list all files and directories, you will see two directory shortcuts:

```
% ls -a
./              ../             alkanes/        animal-counts/ creatures/
numbers.txt     writing/
```

- You've seen `..` which means "the directory above this one"
    - The `.` shortcut means "this current directory

```
% cd .
% pwd
/Users/lpritc/Desktop/shell-lesson-data/exercise-data
```

[SLIDE HERE: Challenge (absolute vs relative paths)]

[SLIDE HERE: Challenge (relative path resolution)]

Syntax of shell commands

- Now that we know something about the `ls` command, and how to move around the filesystem, let's talk about how the commands are put together
    - Let's think about the command `ls -F /`, which lists the contents of the filesystem's root directory.

[SLIDE HERE: shell command syntax]

- `ls` is the *command* - this tells the computer what program runs, or what action to take
- `-F` is an *option* - this changes the behaviour of the command in a particular way
    - options can start with a single dash `-` (short option) or double-dash `--` (long option)
- `/` is an *argument* - it tells the command what it should operate on (usually a file or directory)

- **Commands may take zero, one, or more arguments, and zero, one or more options**
- Each part of the command is separated by a space character.
    - If you miss out the space between `ls` and `-F`, the computer would look for a command called `ls-F`, which doesn't exist
    - **This is why you should never use spaces in filenames** - the computer will misinterpret them
- Note that upper and lower case are different, and this matters

[SHOW SHELL]

```
% ls -s  # displays the size of a file/directory
total 8
0 alkanes/       0 animal-counts/ 0 creatures/     8 numbers.txt    0
writing/
% ls -S  # sorts files/directories by size
alkanes/       creatures/     writing/       animal-counts/ numbers.txt
```

Working With Files and Directories

- We know how to explore files and directories, but how do we *create* them?
- Before we start, let's make sure we're in the `shell-lesson-data` directory

NAVIGATE TO `shell-lesson-data`

```
% pwd
/Users/lpritc/Desktop/shell-lesson-data
```

- Now we'll navigate to the `exercise/writing` directory and see what's in it

```
% cd exercise-data/writing
% ls -F
haiku.txt        LittleWomen.txt
```

Creating directories

- Nelle wants to write her thesis, so let's create a directory to write in
- The command for this is `mkdir` (make directory)
- We can confirm that this created using `ls -F`, and also that there's nothing in it to begin with

```
% mkdir thesis
% ls -F
haiku.txt        LittleWomen.txt  thesis/
% ls -F thesis
```

- We don't have to create only one directory at a time, and we don't have to create a directory only in our current location
- With the `-p` option to mkdir we can create nested subdirectories in a neighbouring directory called `project`
- We can then check the contents of `project` with `ls -FR`

```
% mkdir -p ../project/data ../project/results
% ls -FR ../project
data/    results/

../project/data:

../project/results:
```

[SHOW SLIDE: Names for files and directories]

- Use descriptive filenames so that you don't need to look inside a file to know what it contains
- Stick to lower-case letters, numbers, and these special characters (`, - _`) so you avoid clashes with some special instructions
    - Some filesystems can be case-insensitive so don't distinguish between `thesis` and `Thesis`
- Don't use spaces
- Don't start a name with a dash/hyphen

Create a text file

- Let's create a text file

[SHOW SHELL]

- We'll move our *working directory* to `thesis` and then run a text editor called `Nano` to create a file called `draft.txt`

```
% cd thesis
% nano draft.txt
```

- On my machine this starts an editor called `Pico` - but it's essentially the same thing
- We can type in a couple of lines of text

```
It's not "publish or perish" any more,
it's "share and thrive"
```

- To save the text, we use the `WriteOut` command in `Nano`
    - Hold down the `Ctrl` key, and press `O`

    - We're asked what file name to write the text to - and it's suggesting `draft.txt`
    - Press `Return` to accept this default name

- Once the file is saved, use the `Exit` command to return to the shell
    - Hold down the `Ctrl` key, and press `X`
- We can see that we've saved a file, with `ls`

```
% ls
draft.txt
```

[SHOW SLIDE: File extensions]

- You'll probably have noticed that all of Nelle's files - and most files in general - are called "something dot something"
    - This is just a conventions, we can actually use any name we like
- By convention, file extensions indicate the type of data in the file
    - `.txt` indicates a plain text file
    - `.png` indicates a PNG image
- But this is **only a convention** - the extension might be misleading
    - **And you cannot change file content type by changing the extension**

Moving files and directories

- Let's go back to the `writing` directory

```
% pwd
/Users/lpritc/Desktop/shell-lesson-data/exercise-data/writing/thesis
% cd ..
```

- We might consider that the file we just created has an inappropriate name
    - `draft` is ambiguous, and as it's a collection of quotes, `quotes.txt` might be better
    - So we want to rename `draft.txt` to `quotes.txt`
- To rename a file, we actually need to **move** it, with the command `mv` (short for "move")
    - We need to say what file we're moving, and where we're moving it to

```
% mv thesis/draft.txt thesis/quotes.txt
% ls thesis
quotes.txt
```

- We have to be careful when doing this because `mv` will **silently** overwrite the target file, if it exists
- Suppose we decide that we want to put the `quotes.txt` file in the current director?
    - We use the `mv` (move) command again, but specify the directory we want to move the file to

```
% mv thesis/quotes.txt .
% ls thesis
% ls
haiku.txt        LittleWomen.txt  quotes.txt       thesis/
```

Copying files and directories

- The `cp` command (short for "copy") works just like the `mv` command, except it *copies* a file rather than moving it
    - Let's copy the `quotes.txt` file to `thesis/quotations.txt`

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

- To copy an entire directory and its contents we need to use the `-r` argument (which stands for "recursive")
    - To back up the thesis directory

```
% cp -r thesis thesis_backup
% ls thesis thesis_backup
thesis:
quotations.txt

thesis_backup:
quotations.txt
```

[SLIDE HERE: Challenge (renaming files)]

Removing files and directories

[SHOW SHELL]

- We like the `quotations.txt` file being where it is, so we want to get rid of the redundant `quotes.txt` file
- To do this, we use the `rm` (remove) command

```
% rm quotes.txt
% ls quotes.txt
ls: quotes.txt: No such file or directory
```

- We need to be careful when using `rm`: **There is no trash bin or recovery of deleted files - they're gone forever**

- We can introduce a kind of protection by using the `-i` ("interactive") option, which asks us if we really want to delete a file

```
% rm -i thesis/quotations.txt
remove thesis/quotations.txt? n
```

- We cannot remove directories using `rm` unless we use the `-r` (recursive) option

```
% rm thesis
rm: thesis: is a directory
```

Operations with multiple files and directories

- You've already seen how to give a command multiple filenames, with things like

```
% ls haiku.txt thesis/quotations.txt
haiku.txt              thesis/quotations.txt
```

- Let's move up a directory level to `exercise-data` and look into the `creatures` directory

```
% cd ..
% pwd
/Users/lpritc/Desktop/shell-lesson-data/exercise-data
% ls creatures
basilisk.dat  minotaur.dat  unicorn.dat
```

- We're going to try to copy two of those `.dat` files to a new directory called `backup`

```
% mkdir backup
% cp creatures/minotaur.dat creatures/unicorn.dat backup/
% ls backup
minotaur.dat  unicorn.dat
```

- It's a bit of a pain to write out the filenames in full, so we use **wildcards** - special symbols that represent other characters.
- Let's look at the `alkanes` directory

```
% ls alkanes
cubane.pdb   ethane.pdb   methane.pdb  octane.pdb   pentane.pdb
propane.pdb
```

- It has six files in it, all ending in `.pdb`
- We can use the `*` wildcard to representa any sequence of characters - **including an empty string**
  - The string `*.pdb` represents "any filename ending in `.pdb`
  - But the string `p*.pdb` represents "any filename starting with `p` and ending with `.pdb`

```
% ls alkanes/*.pdb
alkanes/cubane.pdb   alkanes/ethane.pdb   alkanes/methane.pdb
alkanes/octane.pdb   alkanes/pentane.pdb  alkanes/propane.pdb
% ls alkanes/p*.pdb
alkanes/pentane.pdb  alkanes/propane.pdb
```

- The `?` wildcard represents **a single character**

```
% ls alkanes/?ethane.pdb
alkanes/methane.pdb
% ls alkanes/*ethane.pdb
alkanes/ethane.pdb   alkanes/methane.pdb
% ls alkanes/???ane.pdb
alkanes/cubane.pdb  alkanes/ethane.pdb  alkanes/octane.pdb
```

[SHOW SLIDES: Challenge (pattern matching)]

Pipes and Filters

- Now you know how to move around and manipulate the filesystem, we can look at **how to combine existing programs in new ways**
- We'll do this in the `alkanes` directory

[SHOW SHELL]

```
% cd alkanes
% ls
cubane.pdb   ethane.pdb   methane.pdb  octane.pdb   pentane.pdb
propane.pdb
```

- Let's run an example command: `wc` ("word count")

```
% wc cubane.pdb
      20     156    1158 cubane.pdb
```

- This reports the number of *lines* (20), *words* (156), and *characters* (1158) in a file.
- You can use the `*` wildcard to see this information for all files in the directory

```
% wc *.pdb
      20     156    1158 cubane.pdb
      12      84     622 ethane.pdb
       9      57     422 methane.pdb
      30     246    1828 octane.pdb
      21     165    1226 pentane.pdb
      15     111     825 propane.pdb
     107     819    6081 total
```

- We can change the behaviour of `wc` with an option like `-l` to report only the number of lines

```
% wc -l *.pdb
      20 cubane.pdb
      12 ethane.pdb
       9 methane.pdb
      30 octane.pdb
      21 pentane.pdb
      15 propane.pdb
     107 total
```

- Suppose we wanted to know which file was the shortest, in that it contained the fewest lines of text?
- We can see easily enough here with six files (it's `methane.pdb`), but with 6000 that would be more difficult
- **Let's build a tool to do this for us**

Capturing output from commands

- We'll start by **redirecting** the output of `wc` to a file, using the `>` redirection symbol
  - This shows no output on screen because we redirect it to a file

```
% wc -l *.pdb > lengths.txt
% ls lengths.txt
lengths.txt
```

- We can inspect the contents of a file using the `cat` (concatenate) command

```
% cat lengths.txt
      20 cubane.pdb
      12 ethane.pdb
       9 methane.pdb
      30 octane.pdb
      21 pentane.pdb
      15 propane.pdb
     107 total
```

Filtering output

- There is a command called `sort` that sorts the contents of a file
- We can test this out with the `numbers.txt` file in the directory above `alkanes`

```
% cat ../numbers.txt
10
2
19
22
6
% sort ../numbers.txt
10
19
2
22
6
```

- Why do you think the output looks like this?
  - Unless told otherwise, `sort` treats all file data as alphanumerical character strings - i.e. it treats numbers like letters
- To sort numerical data in numerical order, we need to use the `-n` option with `sort`

```
% sort -n ../numbers.txt
2
6
10
19
22
```

- Let's do this with our `lengths.txt` file:

```
% sort -n lengths.txt
       9 methane.pdb
      12 ethane.pdb
      15 propane.pdb
      20 cubane.pdb
      21 pentane.pdb
      30 octane.pdb
     107 total
```

- We can put the sorted list into a new temporary file called `sorted-lengths.txt` using **redirection**
  - Then we can use the `head` command to see only the first line of the file, which gives us the name of the shortest file
  - The `-n 1` option tells `head` that we only want to see one line from the file (`-n 5` would show the first five lines)

```
% sort -n lengths.txt > sorted-lengths.txt
% head -n 1 sorted-lengths.txt
       9 methane.pdb
```

Passing output to another command

- To do this so far, we've had to use two intermediate files: `lengths.txt` and `sorted-lengths.txt`
  - This is an inefficient way to work, and would quickly fill up our filesystem with intermediate files that we'd never use again
- Instead of writing intermediate files, we can **pipe** commands together so that the output of one command becomes the input to the next
  - We do this with the **pipe symbol** `|`
- For instance, to `sort` the contents of `lengths.txt` and use the sorted output as the input to `head -n 1`:

```
% sort -n lengths.txt | head -n 1
       9 methane.pdb
```

- Similarly, we could count the words in all our `.pdb` files and pipe this to `sort`:

```
% wc -l *.pdb | sort -n
       9 methane.pdb
      12 ethane.pdb
      15 propane.pdb
      20 cubane.pdb
      21 pentane.pdb
      30 octane.pdb
     107 total
```

Combining multiple commands

- The magic starts to happen when we realise we don't need to restrict ourselves to only two commands.
- We can chain multiple commands together using pipes:

```
% wc -l *.pdb | sort -n | head -n 1
       9 methane.pdb
```

[SHOW SLIDES: Combining multiple commands AND challenge]*

- This way of linking programs together is one reason why Unix is so successful
  - We don't need to make huge programs that do lots of slightly different things
  - Instead we make tools that do one job well, but that can link well with other small programs

- **You can, and should, write your programs so that they play nicely with existing Unix tools**

Nelle's Pipeline: Checking Files

- Let's get back to Nelle's pipeline
- She's run her samples through the assay machines and has 17 files in the `north-pacific-gyre` directory
    - We'll change directories to see this

[SHOW SHELL]

```
% cd ../../north-pacific-gyre
% ls
goodiff.sh      NENE01729A.txt  NENE01736A.txt  NENE01751B.txt
NENE01843A.txt  NENE01971Z.txt  NENE01978B.txt  NENE02040A.txt
NENE02040Z.txt  NENE02043B.txt
goostats.sh     NENE01729B.txt  NENE01751A.txt  NENE01812A.txt
NENE01843B.txt  NENE01978A.txt  NENE02018B.txt  NENE02040B.txt
NENE02043A.txt
```

- We'll use `wc` on the `.txt` file data to see how large the files are

```
% wc -l *.txt
     300 NENE01729A.txt
     300 NENE01729B.txt
     300 NENE01736A.txt
     300 NENE01751A.txt
     300 NENE01751B.txt
     300 NENE01812A.txt
     300 NENE01843A.txt
     300 NENE01843B.txt
     300 NENE01971Z.txt
     300 NENE01978A.txt
     300 NENE01978B.txt
     240 NENE02018B.txt
     300 NENE02040A.txt
     300 NENE02040B.txt
     300 NENE02040Z.txt
     300 NENE02043A.txt
     300 NENE02043B.txt
    5040 total
% wc -l *.txt | sort -n | head -n 5
     240 NENE02018B.txt
     300 NENE01729A.txt
     300 NENE01729B.txt
     300 NENE01736A.txt
     300 NENE01751A.txt
```

- **We can see there's an issue: one of these files is shorter than all the others**

- When Nelle double-checks the file, she sees that she did the assay first thing on a Monday, and it's possible the machine hadn't been reset.
- Just to be safe, Nelle looks to see if any files are a bit too large

```
% wc -l *.txt | sort -n | tail -n 5
     300 NENE02040B.txt
     300 NENE02040Z.txt
     300 NENE02043A.txt
     300 NENE02043B.txt
    5040 total
```

- Now the sizes are fine, but one of the files appears to have a `Z` in it, which is unexpected
  - Her lab has a convention, though - samples with missing information are labelled with a `Z`
  - So she checks for more of them

```
% ls *Z.txt
NENE01971Z.txt  NENE02040Z.txt
```

- When Nelle checks the logs, she sees that no depth information was recorded for these files, so they would need to be excluded from her analysis

Loops

- Nelle wants to run the `goostats.sh` program on her data files, but it will take her a long time to do so if she has to type the command in each time.
  - All that typing is also prone to error - it's easy to become bored or distracted
- She can solve this problem if she knows about **loops**
  - **Loops** let us repeat a command, or set of commands, on multiple inputs - and they are key to automating analyses
- Let's move to the `exercise-data/creatures` directory:

```
% cd ../exercise-data/creatures
% ls
basilisk.dat  minotaur.dat  unicorn.dat
```

- There are a set of `.dat` files.
- Let's look at their contents with `head`

```
% head -n 5 *.dat
==> basilisk.dat <==
COMMON NAME: basilisk
```

```
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
CCCCAACGAG
GAAACAGATC

==> minotaur.dat <==
COMMON NAME: minotaur
CLASSIFICATION: bos hominus
UPDATED: 1765-02-17
CCCGAAGGAC
CGACATCTCT

==> unicorn.dat <==
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
AGCCGGGTCG
CTTTACCTTA
```

- All of these files have the same structure
  - On line two there is a `CLASSIFICATION` for each creature
  - Now suppose we wanted to print out the classification for each species
    - We could do this by combining `head -n 2` and `tail -n 1`

```
% head -n 2 basilisk.dat | tail -n 1
CLASSIFICATION: basiliscus vulgaris
```

- Doing this multiple times is a pain, so we would turn to loops to solve the problem

[SHOW SLIDE: The structure of a loop - ANIMATED]

- The general form of a loop is shown on-screen
- What we want to do is operate on each **thing** in a **list of things**
- We enclose what we want to do in a "job execution list"
  - between **do** (start of the loop)
  - and **done** (end of the loop)
- Between **do** and **done** we place the commands that we want to apply to each **thing**
- **Note that there is a specific change to the syntax here - `thing` has a `$` in front of it, indicating that it is a **variable**
- So let's try this for our example
  - Note that, as we hit "Return" on each line, we get a new prompt style to indicate we haven't finished typing the command/closed the for loop, yet

[SHOW SHELL]

```
% for filename in basilisk.dat minotaur.dat unicorn.dat
for> do
for>     echo $filename
for>     head -n 2 $filename | tail -n 1
for> done
basilisk.dat
CLASSIFICATION: basiliscus vulgaris
minotaur.dat
CLASSIFICATION: bos hominus
unicorn.dat
CLASSIFICATION: equus monoceros
```

- We have started using something called a **variable**
    - This is essentially a "box" with a name on it (the *variable name*)
    - Each time we go round the loop, the box gets the next item in the list of things
    - Within the loop, rather than referring to the *thing* itself, we refer to the box that contains it
- In our loop, the box has the name `filename`, and we refer to it with the `$` to indicate that it is a variable: `$filename`
- As the loop proceeds, the "filename" box first contains `basilisk.dat`, `minotaur.dat`, and finally `unicorn.dat`
- To some extent the variable name itself doesn't matter
    - But it is good practice to **make your code readable by using a desriptive variable name**
- Using `x` works just as well as `filename`, but `filename` is clearer to understand

```
% for x in basilisk.dat minotaur.dat unicorn.dat
for> do
for>     head -n 2 $x | tail -n 1
for> done
CLASSIFICATION: basiliscus vulgaris
CLASSIFICATION: bos hominus
CLASSIFICATION: equus monoceros
```

[SHOW SLIDE: Challenge (write your own loop)]

[SHOW SHELL]

- We can use wildcards in loops

```
% for filename in *.dat
for> do
```

```
for>     head -n 2 $filename | tail -n 1
for> done
basilisk.dat
CLASSIFICATION: basiliscus vulgaris
minotaur.dat
CLASSIFICATION: bos hominus
unicorn.dat
CLASSIFICATION: equus monoceros
```

- We can also use another redirection symbol: `>>` to **append** data to a file (which we then inspect with `cat`)

```
% for filename in *.dat
for> do
for>     head -n 2 $filename | tail -n 1 >> classification.txt
for> done
% cat classification.txt
basilisk.dat
CLASSIFICATION: basiliscus vulgaris
minotaur.dat
CLASSIFICATION: bos hominus
unicorn.dat
CLASSIFICATION: equus monoceros
```

Nelle's pipeline: processing files

- We now know enough for Nelle to process her data files using `goostats.sh`
- This is a program written by her supervisor that calculates some statistics
    - It takes two inputs:
        - An input file (the raw data)
        - An output file (where the statistics will be stored)
- Let's go back to Nelle's data directory and make sure we can apply the program to the files we want
    - This is only the files ending in `A` or `B` as they don't have missing data

```
% cd ../../north-pacific-gyre
% for datafile in NENE*A.txt NENE*B.txt
for> do
for> echo $datafile
for> done
NENE01729A.txt
NENE01736A.txt
NENE01751A.txt
NENE01812A.txt
NENE01843A.txt
NENE01978A.txt
NENE02040A.txt
NENE02043A.txt
NENE01729B.txt
```

```
NENE01751B.txt
NENE01843B.txt
NENE01978B.txt
NENE02018B.txt
NENE02040B.txt
NENE02043B.txt
```

- So we can loop over all the correct files - that's good
- Now we need to decide what to call the output files for `goostats.sh`
    - Nelle decides to prepend the string `stats`

```
% for datafile in NENE*A.txt NENE*B.txt
for> do
for> echo $datafile stats-$datafile
for> done
NENE01729A.txt stats-NENE01729A.txt
NENE01736A.txt stats-NENE01736A.txt
NENE01751A.txt stats-NENE01751A.txt
NENE01812A.txt stats-NENE01812A.txt
NENE01843A.txt stats-NENE01843A.txt
NENE01978A.txt stats-NENE01978A.txt
NENE02040A.txt stats-NENE02040A.txt
NENE02043A.txt stats-NENE02043A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01751B.txt stats-NENE01751B.txt
NENE01843B.txt stats-NENE01843B.txt
NENE01978B.txt stats-NENE01978B.txt
NENE02018B.txt stats-NENE02018B.txt
NENE02040B.txt stats-NENE02040B.txt
NENE02043B.txt stats-NENE02043B.txt
```

- So we have all the arguments we need
- Now we can replace `echo` with `bash goostats.sh` (which is how we run the program)

```
% for datafile in NENE*A.txt NENE*B.txt
for> do
for> echo $datafile stats-$datafile
for> bash goostats.sh $datafile stats-$datafile
for> done
NENE01729A.txt stats-NENE01729A.txt
NENE01736A.txt stats-NENE01736A.txt
NENE01751A.txt stats-NENE01751A.txt
NENE01812A.txt stats-NENE01812A.txt
NENE01843A.txt stats-NENE01843A.txt
NENE01978A.txt stats-NENE01978A.txt
NENE02040A.txt stats-NENE02040A.txt
NENE02043A.txt stats-NENE02043A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01751B.txt stats-NENE01751B.txt
```

```
NENE01843B.txt stats-NENE01843B.txt
NENE01978B.txt stats-NENE01978B.txt
NENE02018B.txt stats-NENE02018B.txt
NENE02040B.txt stats-NENE02040B.txt
NENE02043B.txt stats-NENE02043B.txt
```

Shell scripts

- Now we're going to see what makes the shell such a powerful environment for data analysis and computational science
- We're going to take the commands we repeat frequently and save them in files so that we can re-run them on new data by typing a single command
- These files are called **shell scripts**, and they are small **programs**.
- Using scripts makes your work faster and more efficient
    - It also makes it more accurate and less prone to errors through typing/copy-paste
    - It also makes your work more reproducible - you can share scripts with colleagues and reviewers (and get the credit!)
- Let's go back to `alkanes` and create a new file called `middle.sh` - this will be our shell script

```
% cd ../exercise-data/alkanes
% nano middle.sh
```

- Now we are going to write a shell command in the new file

```
head -n 15 octane.pdb | tail -n 5
```

- This is a variation on the pipe we made earlier
    - This command selects the first 15 lines of `octane.pdb` and then selects the last 5 of those 15 lines
- Then save the file (`Ctrl+O`) and exit (`Ctrl-X`)
- To run this file as a shell script, we use the command `bash middle.sh`

```
% bash middle.sh
ATOM      9  H           1      -4.502   0.681   0.785  1.00  0.00
ATOM     10  H           1      -5.254  -0.243  -0.537  1.00  0.00
ATOM     11  H           1      -4.357   1.252  -0.895  1.00  0.00
ATOM     12  H           1      -3.009  -0.741  -1.467  1.00  0.00
ATOM     13  H           1      -3.172  -1.337   0.206  1.00  0.00
```

- This is giving us exactly what we'd get if we ran the command in `middle.sh` directly
- But what if we wanted to perform the same action on a different file?
    - `octane.pdb is "hard-coded" into the script
- Shell scripts allow us to use special variables to catch filenames
- Let's open `middle.sh` again and do that

```
% nano middle.sh
```

```
head -n 15 "$1" | tail -n 5
```

- The `"$1"` tells the script to use the first argument that comes after the command to run the script.
- Let's run our script specifying the `octane.pdb` file as input

```
% bash middle.sh octane.pdb
ATOM      9  H           1      -4.502    0.681    0.785  1.00  0.00
ATOM     10  H           1      -5.254   -0.243   -0.537  1.00  0.00
ATOM     11  H           1      -4.357    1.252   -0.895  1.00  0.00
ATOM     12  H           1      -3.009   -0.741   -1.467  1.00  0.00
ATOM     13  H           1      -3.172   -1.337    0.206  1.00  0.00
```

- We get the same result as last time
- But what if we try it on `pentane.pdb`?

```
% bash middle.sh pentane.pdb
ATOM      9  H           1       1.324    0.350   -1.332  1.00  0.00
ATOM     10  H           1       1.271    1.378    0.122  1.00  0.00
ATOM     11  H           1      -0.074   -0.384    1.288  1.00  0.00
ATOM     12  H           1      -0.048   -1.362   -0.205  1.00  0.00
ATOM     13  H           1      -1.183    0.500   -1.412  1.00  0.00
```

- We can now run this program/command on any file we like

More arguments

- But what if we want to change the range of lines that get returned?
- We can use special variables `"$2"` and `"$3"` (and so on) to collect the second and third command line arguments
- Let's use these to specify the line numbers we want to return
    - So that `middle.sh filename 20 5` will report lines 15 to 20 from `filename`
- We open `middle.sh` in Nano again

```
% nano middle.sh
```

```
head -n "$2" "$1" | tail -n "$3"
```

- Let's run this on `pentane.pdb`

```
% bash middle.sh pentane.pdb 15 5
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
% bash middle.sh pentane.pdb 20 5
ATOM     14  H           1      -1.259   1.420   0.112  1.00  0.00
ATOM     15  H           1      -2.608  -0.407   1.130  1.00  0.00
ATOM     16  H           1      -2.540  -1.303  -0.404  1.00  0.00
ATOM     17  H           1      -3.393   0.254  -0.321  1.00  0.00
TER      18              1
```

Commenting a script

- This works, and it's great that we understand what's going on
- But in six months time we might forget what this script is doing, and a new person might take a while to work it out
    - So we add comments to our script

```
% nano middle.sh
```

```
# Select lines from the middle of a file.
# Usage: bash middle.sh filename end_line num_lines
head -n "$2" "$1" | tail -n "$3"
```

- The `#` character means that Bash doesn't read the line as an instruction, but instead just as a comment
- **It is good practice to leave sufficient comments so that you in the future (and others) can understand your scripts**

Using wildcard arguments

- Now what if we want to process a lot of files at the same time?
- Suppose we want to sort all our `.pdb` files by length?
    - At the terminal we would type:

```
% wc -l *.pdb | sort -n
       9 methane.pdb
      12 ethane.pdb
      15 propane.pdb
      20 cubane.pdb
      21 pentane.pdb
      30 octane.pdb
     107 total
```

- If we wanted to get all those filenames into the script we couldn't use `$1`, `$2`, and so on because we don't know how many filenames there would be.
- Instead, we use the special variable `$@` which means "all of the arguments to the script"
- We'll create a new file called `sorted.sh`

```
% nano middle.sh
```

```
# Sort files by their length.
# Usage: bash sorted.sh one_or_more_filenames
wc -l "$@" | sort -n
```

- Now we can run this on all the `.pdb` files in the current directory

```
% bash sorted.sh *.pdb
       9 methane.pdb
      12 ethane.pdb
      15 propane.pdb
      20 cubane.pdb
      21 pentane.pdb
      30 octane.pdb
     107 total
```

- We can also run it on arbitrary collections of files

```
% bash sorted.sh *.pdb ../creatures/*.dat
      12 ethane.pdb
      15 propane.pdb
      20 cubane.pdb
      21 pentane.pdb
      30 octane.pdb
     163 ../creatures/basilisk.dat
     163 ../creatures/minotaur.dat
     163 ../creatures/unicorn.dat
     596 total
```

Nelle's pipeline script

- Back with Nelle, her supervisor insists - rightly - that her analysis needs to be reproducible and that she should use a script
- Let's go back to her project and create a new script called `do-stats.sh`

```
% cd ../../north-pacific-gyre
% nano do-stats.sh
```

```
# Calculate stats for data files.
for datafile in "$@"
do
    echo $datafile
    bash goostats.sh $datafile stats-$datafile
done
```

- Nelle can now run this script on her files with a single command:

```
% bash do-stats.sh NENE*A.txt NENE*B.txt
```

Summary

- We've gone through quite a lot, and it might take some time to digest everything
- But here's a summary of some of the topics we've been through
    - running commands at the terminal/command-line prompt
        - using command line options and arguments
    - finding the current working directory (`pwd`) and listing its contents (`ls`)
    - navigating around the filesystem with `cd`
    - creating, copying, moving, and deleting directories and files with `mkdir`, `cp`, `mv`, `rm`
    - understanding file extensions
    - using shell commands `wc`, `sort`, `head`, `tail`
    - redirecting output with `>` and `>>`
    - piping program output with `|` and chaining arbitrary commands together
    - writing loops to automate and iterate analyses over multiple files
    - combining all of this into shell scripts for reproducibility and flexibility
- **PLEASE WRITE ONE THING YOU LIKED/WE DID WELL ON THE GREEN STICKY; ONE THING WE COULD IMPROVE ON THE RED STICKY**