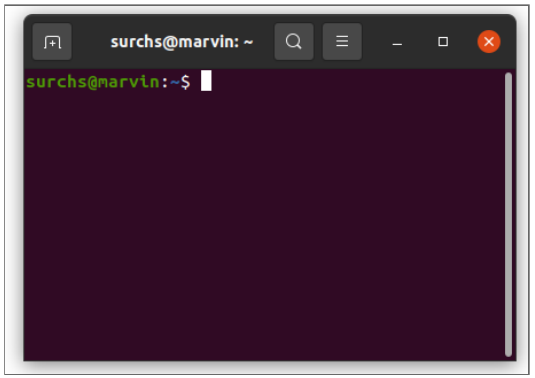
Introduction to the bash shell - exercises

- A brief lecture recap (p.5-23)
- Exercises (p.24 onwards)

Setting up for the exercises

1. Open your terminal



1. and confirm that you are running the **bash** shell:

In [ ]:
```
echo $SHELL
# /bin/bash
```

1. **Copy** the data for the exercises (`shell-course`) out of the **QLS-course-materials** directory you previously cloned and into your **home directory** (e.g. `/home/USERNAME` or `/Users/USERNAME`):

```
In [ ]: cp -r ~/QLS-course-materials/Lectures/2025/02_terminal-bash/shell-course ~
```

Replace `~` with the path where you stored `QLS-course-materials` (if not your home directory)

A brief recap of the lecture

1. How to **navigate** files and directories
2. How to **modify and move** files and directories
3. How to **find** files and directories
4. Shell **variables** and scripts

Examples are provided in code snippets for you to try in your terminal as desired.

Bash command recap

```
In [ ]: ls -l ~/shell-course
```

A shell command has 3 parts:

1. A **command** (`ls`),
2. One or more **options** (`-F`), also called a **flag** or a **switch**, and
3. An **argument** (`~/shell-course`)

It is generally recommended to follow the order: \<command> [**OPTIONS**] <_argument_> (e.g., head **-n 1** *myfile.txt*)
However, depending on your OS and versions of installed commands, the order of options and arguments might be less strictly enforced.
For example, on some machines both these forms will work:

```
head -n 1 myfile.txt
head myfile.txt -n 1
```

When in doubt, always specify options first, arguments second.

Recap: navigation

- The file system is responsible for managing information on the disk
- Directories can also store other (sub-)directories, which forms a directory tree
- `cd <path>` changes the current working directory
- `ls <path>` prints a listing of a specific file or directory; `ls` on its own lists the current working directory
- `pwd` prints the user's current working directory
- `/` on its own is the root directory of the whole file system
- A relative path specifies a location starting from the current location
- An absolute path specifies a location from the root of the file system
- `..` means "the directory above the current one"; `.` on its own means "the current directory"

Refresher `pwd`

`pwd` -> print working directory

```
In [ ]: pwd
```

- `pwd` lets you know where you are
- It always prints absolute paths
- It's a great way to quickly get your bearings

Refresher `ls`

`ls` -> list directory contents

```
In [ ]: ls -la
```

- `ls` lists the contents of the current working directory by default
- you can give it many options to change what is printed
- you can list other directories, by supplying them as arguments
- `.` stands for the current working directory
- `..` stands for the parent directory, the directory above the current directory
- file/directory names beginning with `.` are hidden from listing by default (here, the `-a` (`--all`) flag displays them)

Refresher `cd`

`cd` -> changes your working directory

```
In [ ]: pwd
```

```
In [ ]: cd ~/shell-course
```

```
In [ ]: pwd
```

- `cd` changes the current working directory
- it expects a relative or absolute path to the new working directory you want to change to
- if you give it no argument, it will go back to your home directory

Refresher `home directory`

- Your home directory is where your user-specific files are
- on Linux it is in `/home/USERNAME`
- on Mac it is in `/Users/USERNAME`
- it contains your files and config files

```
In [ ]: pwd
```

```
In [ ]: cd
```

```
In [ ]: pwd
```

```
In [ ]: cd -
```

- `cd` without arguments brings you to your home directory
- `~` is a shorthand for your home directory -> `cd ~` also brings you there
- your home directory has a path -> `cd /home/USERNAME` also brings you there
- `-` is a shorthand for the directory you were in before the last `cd` call

Recap: modifying things

- `cp <old> <new>` copies a file (`cp -r` for a directory)
- `mkdir <path>` creates a new directory
- `mv <old> <new>` moves (renames) a file or directory
- `rm <path>` removes (deletes) a file (`rm -r` for a directory)
- `touch <file>` creates an empty text file or updates the access time of an existing file
- `*` matches zero or more characters in a filename, e.g. `*.txt` matches `a.txt` and `any.txt` (all files ending in `.txt`)
- `?` matches any single character in a filename, e.g. `?.txt` matches `a.txt` but not `any.txt`
- The shell does not have a trash bin: once something is deleted, it's really gone

Refresher `cp`

`cp` -> copy files and directories to a new path

```
In [ ]: cd ~/shell-course/interesting_files
```

```
In [ ]: ls
```

```
In [ ]: cp the_meaning_of_life.txt the_meaning_of_life_backup.txt
```

```
In [ ]: ls
```

- `cp` (and `mv` too) takes the form `cp [old-path] [new-path]`
- `cp` can operate on many files at once as long as the target is a directory (need the `-r` flag!)
- `cp` will keep the original file, whereas `mv` will move it, i.e. destroying the original
- `cp` and `mv` will overwrite without asking -> **dangerous**. The `-i` flag will make them ask first

Refresher `rm`

`rm` -> removes files and directories

```
In [ ]: ls
```

```
In [ ]: rm the_meaning_of_life_backup.txt
```

```
In [ ]: ls
```

- `rm` generally deletes files without first asking
- `rm` deletes things **forever**. There is no trash-bin for bash and no undo button
- `rm` cannot delete directories without the `-r` flag

Recap: finding things

- we can print the structure of any given directory with `tree` (or `ls -R`)
- `find` is a great tool to search for **files and directories** based on their name and other meta-data like size, age, and so on
- `grep` is a great tool to search **within (text)files** for occurrences of a given string or even complex regular expressions
- pipes (`|`) allow us to combine the output of one command with the input of another command

Refresher `find`

`find` -> find files and directories by name and meta-data

```
In [ ]: cd ~/shell-course
        find . -name "my*"
```

- `find` is great to find all files with a certain name pattern
- `find` can also search for attributes like size and age

Refresher `grep`

`grep` -> find a text pattern inside of files and print the matches

```
In [ ]: grep "rabbit" flying_circus/* --max-count 2
```

- `grep` is great to search for something **inside** of files
- `grep` can search for a simple string or complex regular expressions
- `grep` can be useful to extract lines with specific content out of a file

Refresher pipes

The `|` character is a pipe. It can be used to link the output of one bash command to the input of another bash command. Commands linked in this way are called pipelines

```
In [ ]: grep "rabbit" flying_circus/*.txt --max-count 2 | wc --chars
```

- shell commands generally do one thing well
- linking commands can achieve powerful pipelines
- here `grep` finds text files and pipes the output to `wc` a program to count the number of characters and lines in a text
- here we then get the total number of characters found by `grep`
- `>` and `>>` are special characters that can redirect output into files (we'll see this in a moment)

Refresher help

The bash shell has many great helper tools. Often they can answer questions without the need for google:

- `--help` -> a flag that provides the basic usage and options for many bash commands
- `whatis` -> provides a brief description of a command
- `man` -> opens the manual for a given command, with comprehensive documentation of functionality, options, and usage examples
- `which` -> tells you where the program is located that is called by a command

In [ ]:
```
whatis wc
```

In [ ]:
```
wc --help
```

In [ ]:
```
which wc
```

Recap: variables and scripts

- the `$PATH` variable defines the directories where the shell will look for commands
- you can change `$PATH` for just your current shell session, or for all sessions (in your `~/.bashrc` file - do this with caution!)
- the `$` character is necessary to refer to the **value** of a bash variable
- often it makes sense to put the variable name inside curly braces `${ }` when referencing the value, to differentiate it from other text
- variables must be exported to be made accessible inside other scripts/programs: `export VARIABLE_NAME`
- shell scripts are executable text files that contain shell commands
- scripts need execution permission that we can give with the `chmod` command
- scripts start with the "shebang": `#!/bin/bash` that specifies which shell the script should be interpreted by
- scripts are great to document what you did or do it again many times

In [ ]:
```
echo ${PATH}
```

In [ ]:
```
which grep
```

Recap of topics

1. How to **navigate** files and directories (`ls`, `cd`, `pwd`)
2. How to **modify and move** files and directories (`cp`, `mv`, `rm`)
3. How to **find** files, directories and help (`find`, `grep`, `tree` and `man`, `whatis`, `which`)
4. Shell **variables** and scripts (`$PATH`, `.bashrc` , `${MY_VAR}`, `export`)

Exercise 1 - moving things around

Let's navigate to the `dir_of_doom` with `cd` and take a look inside with `ls` and `tree`.

In [ ]:
```
cd ~/shell-course/dir_of_doom
```

In [ ]:
```
tree
```

In [ ]:
```
ls -Rla
```

All these files are in `the_wrong_dir`, we want to move them to `the_right_dir` using wildcards so we don't have to move each file separately.

Exercise 1a

Use the `mv` command to move all the files from `the_wrong_dir` to `the_right_dir`. Remember the form of the `mv` and `cp` commands: `mv [old_path] [new_path]`.

Hints

- the `mv` command can move many files at once, as long as the `[new_path]` is a directory and not a file
- e.g., `mv file1.txt file2.txt target_directory` works, but `mv file1.txt file2.txt file3.txt` does not
- a wildcard expands to match multiple file names. It has the same function as typing all the file names by hand
- `*` (asterisk) will match any character 0 or more times
- `?` (question mark) will match any character exactly once

Try it out!

In [ ]:
```
mv the_wrong_dir/my_file?.txt the_right_dir
```

In [ ]:
```
# Check that it has worked
tree
```

Exercise 1b

Now that everything in `the_right_dir` looks good, remove the `the_wrong_dir`.
Try it out!

Remember:

- `rm` can remove files
- `rm` can only remove directories with the `-r` ("recursive") flag

```
In [ ]: ls -la the_wrong_dir
```

```
In [ ]: rm -r the_wrong_dir
```

Now `the_wrong_dir` is gone and there is no way to get it or its contents back! Be **very careful** with `rm` commands, especially when using wildcards and relative paths

Summary

- `cp old new` copies a file
- `mkdir path` creates a new directory
- `mv old new` moves (renames) a file or directory
- `rm path` removes (deletes) a file
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt`
- `?` matches any single character in a filename, so `?.txt` matches `a.txt` but not `any.txt`
- The shell does not have a trash bin: once something is deleted, it's really gone

Exercise 2 - pipes

Now let's take a look in the `flying_circus` directory. There we have several text files and we want to know what the shortest text file is. Here we can make use of several tools:

- `wc`
- `sort`
- `head`

Use the `whatis` command to find out what they do.

In [ ]:
```
whatis head
```

Exercise 2a

Navigate to the `flying_circus` directory, and print the **number of lines** of each text file in the directory using a single `wc` command.

Hints

- A wildcard (`*`) expands to match multiple file names
- The `--help` flag shows all options you can use to control the behaviour of a command

Try it out!

In [ ]:
```
ls -lF
```

In [ ]:
```
wc -l *.txt
```

Exercise 2b

Instead of printing the `wc` command output to the screen (called STDOUT), redirect the output into a file `file_length.txt`.

Hints

Some special characters that redirect the output normally printed to STDOUT:

- `|` ("pipe") redirects the output to a second bash command as input
- `>` redirects the output to a file and **overwrites** whatever is in the file
- `>>` redirects the output to a file and **appends** to this file if it exists

Try it out!

In [ ]:
```
wc -l *.txt > file_length.txt
```

In [ ]:
```
ls
```

In [ ]:
```
cat file_length.txt
```

Now let's sort the text in `file_length.txt` by the number of lines with `sort`:

```
In [ ]: sort file_length.txt
```

Notice the file lengths have not been sorted correctly. `sort` interpreted the numbers as text, but we want them interpreted as numbers. From `man sort`, we know that `--numeric-sort` achieves this behaviour. Try redirecting the correct, numerically-sorted output into a file called `sorted_length.txt`.
Note: Newer versions of `sort` may have desired behaviour without the `--numeric-sort` flag, due to how the STDOUT of `wc -l *.txt` is formatted. If this is the case, try `--numeric-sort` on the contents of `dangerous_rabbits.txt` instead.

```
In [ ]: sort --numeric-sort file_length.txt > sorted_length.txt
```

Now let's read only the first line of `sorted_length.txt` to find the name of the shortest text file in the `flying_circus` directory.

```
In [ ]: head -n 1 sorted_length.txt
```

Exercise 2c

So far, to find the shortest text file we have run:

1. `wc -l *.txt > file_length.txt`
2. `sort --numeric-sort file_length.txt > sorted_length.txt`
3. `head  -n 1 sorted_length.txt`

This created 2 text files we didn't really care about and took 3 commands. This is a good use for bash pipelines!

Remember: the `|` (pipe) character redirects the output to another command as input.

Try to rewrite the 3 commands into one pipeline command using `|`, so that the name of the shortest file is printed without creating any files.

Try it out!

In [ ]:
```
wc -l *.txt | sort --numeric-sort | head -n 1
```

Summary

- `|` the "pipe" character redirects the output to a second bash command as input. e.g. `wc -l *.txt | head -n 1`
- `>` redirects the output to a file and **overwrites** whatever is in the file. e.g. `wc -l *.txt > file_lengths.txt`
- `>>` redirects the output to a file and **appends** to this file if it exists. e.g. `wc -l *.txt >> file_lengths.txt`

Exercise 3 - grep

Some of the text files in the `flying_circus` directory are so long because they contain the complete scripts to movies from Monty Python (which `python` is named after!). For example, `brian.txt` contains the script to "The Life of Brian". Let's say our goal is to make a personalized script for the actor playing the role of `"Brian"` - containing only the lines said by the role.
For this we can use the tool `grep`. `grep` can search for text snippets (i.e. strings) **inside** of files. We can redirect the output of `grep` into new text files. Let's first do this for `"Brian"`:

In [ ]:
```
ls
```

In [ ]:
```
grep "Brian" brian.txt
```

Now we get every string containing `"Brian"`. But we only want the strings that are lines the character Brian says. We can do two things:

- search specifically for the string `"Brian:"` including the `:` character
- use the `^` (caret) character to only find occurrences that are **at the beginning of the line**, i.e. `^Brian:`
    - FYI: the `man` pages for `grep` have a section dedicated to these patterns, called regular expressions

Let's add these to our grep command and then redirect the output to a file in `my_lines/Brians_lines.txt`

```
In [ ]: mkdir my_lines
```

```
In [ ]: grep "^Brian:" brian.txt > my_lines/Brians_lines.txt
```

```
In [ ]: head my_lines/Brians_lines.txt
```

That's nice. But is there an easy way to:

- create the lines for another role in this movie?
- record the exact command we used to create the lines for this role?
- re-run the exact same command in the future, e.g. to re-create the lines for the "Brian" role?
- be able to change the role we create lines for without modifying commands?

Exercise 3 - scripts

We can use shell scripts! Shell scripts are just special text files that contain shell commands. We can

1. take the commands we have just written and put them in a shell script to re-run them again later.
2. use a variable to store the name of the role so we can easily change what actor we generate lines for

Let's look at these steps separately.

Recap scripts

A **script** is a text file that contains shell commands and:

1. commonly has the `.sh` file ending to show it is a script
2. has execution permission. This can be given with the `chmod +x` command
3. starts with the shebang: `#!/bin/bash` that specifies the shell that should run the script

Let's look at an example script in the `interesting_files` directory:

In [ ]:
```
ls -lF ../interesting_files/
```

In [ ]:
```
../interesting_files/run_me.sh
```

Exercise 3a

Create a script that runs our `grep` command to create the lines for the role of "Brian"
Steps:

1. In the `~/shell-directory/flying_circus` directory, create an empty script file called `create_lines.sh`
2. Copy the `grep` command to the shell script using a text editor (or, try using `echo` and `>` to redirect the entire command into the file)
3. Add the necessary shell script elements:
    A. a `.sh` file ending (done)
    B. first line has the shebang: `#!/bin/bash`
    C. file has execution permission. This can be given with the `chmod +x` command

```
In [ ]: grep "^Brian:" brian.txt > my_lines/Brians_lines.txt
```

Try it out!.

HINTS:

- `touch <filename>` creates an empty file
- if you copy by hand, use the context menu to paste (right-click). `CTRL+C` is reserved in `bash` to cancel commands
- in `nano`, remember to save (write out) the file before you exit.
    - `^` for CTRL: `^G` means "press and hold CTRL together with the `G` key"
    - `M` for ALT: `M-U` means "press and hold ALT together with the `U` key"
    - Write out then is CTRL+`O`

In [ ]:
```
nano create_lines.sh

# OR
echo \
  'grep "^Brian:" brian.txt > my_lines/Brians_lines.txt' > create_lines.sh
# and then nano create_lines.sh to add shell script elements
```

Now let's

- give the script execution permission with `chmod +x`
- and see if this worked with `ls -lF` (executable files are marked with `*`)
- finally, run our script with `./create_lines.sh` (remove the existing `my_lines/Brians_lines.txt` file first)

In [ ]:
```
ls -lF
```

In [ ]:
```
chmod +x create_lines.sh
```

In [ ]:
```
ls -lF
```

In [ ]:
```
rm my_lines/Brians_lines.txt
./create_lines.sh
```

Exercise 3 - variables

But what if we want to change the role that the script creates the lines for? We can use variables!

Recap variables

We can define **variables** and assign values to them

- to define a variable we use the `=` character
- to access the value of a variable we use the `$` character
- a newly defined variable is a **shell variable** that is not visible to other programs we start from our shell
- with the `export` command, we can turn our variable into an **environment variable** that is visible to other programs

Exercise 3b

1. Create a shell variable called `ROLE` and assign the name of a different role, `"Vendor"`, as the value
2. Confirm the value was correctly assigned with `echo`
3. Turn our variable `ROLE` into an **environment variable** so our script can see it
4. Confirm that `ROLE` is now part of the environment variable list using `printenv` and `grep`

Try it out!.

```
In [ ]: ROLE="Vendor"
```

```
In [ ]: echo ${ROLE}
```

```
In [ ]: export ROLE
```

```
In [ ]: printenv | grep ROLE
```

Exercise 3c

Finally, replace the hard-coded value `"Brian"` in the script `create_lines.sh` with the new variable `ROLE`.
To confirm it worked, try changing the `ROLE` variable in the shell to other roles from the script and then run `create_lines.sh` again. Some other roles to try:

- Baby
- Balthasar
- Eremite
- Door

Hints

- `$` to access the value of the variable
- variable names are case sensitive
- wrapping the variable in `{ }` is recommended when the variable is surrounded by other text, so bash knows where the variable name ends
    - we can embed a variable in a string like this: `"Hello World!"` -> `"Hello ${PLACE}!"`

Try it out!

In [ ]:
```
cat create_lines.sh
```

In [ ]:
```
./create_lines.sh
        # old: grep "^Brian:" brian.txt > my_lines/brians_lines.txt
 # new: grep "^${ROLE}:" brian.txt > my_lines/${ROLE}s_lines.txt
```

In [ ]:
```
ROLE=Door
```

In [ ]:
```
./create_lines.sh
```

Summary

- `grep` is a great tool to search within (text)files for occurrences of a given string or even complex regular expressions
- shell scripts are a very powerful way to automate, repeat and document steps
- variables can store values that scripts operate on
- we access the value of variables with `$` and we can export variables to environment variables with `export`
- here we knew which variables were used inside the script and changed those in the environment. In practice, there are better ways to tell our script which actor we want to have lines created for (i.e. we can make our script accept its own arguments like other bash commands)

Exercise 4 - a neuroimaging dataset example

When working with research datasets, commands like `head`, `tail`, `cat`, `wc`, and `grep` are extremely useful to quickly check info or get an overview of large tabular data files (e.g., `.csv` and `.tsv`) without having to load them in Excel, Google Sheets, etc.
The file `participants_nbsub-100.tsv` is a tab-separated table that contains data for 100 subjects from the ABIDE dataset. Each row represents a subject except for the first row, which contains the column names.

In [ ]:
```
cd ~/shell-course
```

In [ ]:
```
ls
```

Exercise 4

Explore `participants_nbsub-100.tsv`, **without opening the file**:

1. Print only the first row (the column names) to see what kinds of subject data are available
2. Confirm that besides the header, there are 100 rows (lines of subject data) in the file
3. Print just the information for the subject `50012`
4. There's a column `SITE_ID` which has 3 possible values: `PITT`, `OLIN`, `OHSU`. Find just the rows that have the value `"PITT"`, and output those rows to a new file `participants_PITTonly.tsv`

Try it out!

```
In [ ]: head -n 1 participants_nbsub-100.tsv
```

```
In [ ]: tail -n +2 participants_nbsub-100.tsv | wc -l
```

```
In [ ]: grep 50012 participants_nbsub-100.tsv
```

```
In [ ]: grep PITT participants_nbsub-100.tsv > participants_PITTonly.tsv
```

Exercise 5 - the `$PATH` variable

When we run scripts that we have created (like `create_lines.sh`), we need to specify the path to the script: `./create_lines.sh` (remember that `.` stands for the current directory).
From the lecture we know that when you type a command into the shell, it will go and search for executable files with this name in a number of directories. These directories are defined in the `$PATH` variable:

```
In [ ]: echo $PATH
```

Unless a script is in one of these directories, the shell won't find it. We have some scripts inside of the `interesting_files` directory, but the directory is not in the `PATH` variable. We shouldn't be able to run them without specifying their path:

```
In [ ]: pwd
```

```
In [ ]: ls -lF interesting_files
```

```
In [ ]: run_me.sh
```

Exercise 5a

How can we add the `interesting_files` directory to the `PATH` variable?
Just like the `ROLE` variable in the previous exercise, we can **re-assign** the value of the `PATH` variable.

Hints:

- In `$PATH`, directories are separated by a `:` character
- Append another directory to the list of directories currently in `$PATH` using the `:` delimiter, and re-assigning the combined path to the `PATH` variable

Try it out!

In [ ]:
```
echo $PATH
```

In [ ]:
```
PATH=${PATH}:~/shell-course/interesting_files
```

In [ ]:
```
echo $PATH
```

And now let's see if the shell can find our script

```
In [ ]: pwd
```

```
In [ ]: run_me.sh
```

Note that modifying `$PATH` using `export` only affects the current shell session! To make this permanent, we would need to make the change in `~/.bashrc`.

Exercise 5b

Create a new directory `~/shell-course/my_scripts`, and create a script inside called `favorite_color.sh` that prints to STDOUT:

```
My favourite color is: <FAVORITE COLOR>!
```

where `<FAVORITE COLOR>` can be set by an environment variable.
Then, make the necessary changes to `$PATH` so that the script can be run just by calling the filename: `favorite_color.sh`

Try it out!

Hints

- You can use single quotes `''` to prevent the Bash from interpreting special characters in a string

In [ ]:
```
mkdir ~/shell-course/my_scripts
cd ~/shell-course/my_scripts
```

In [ ]:
```
echo 'echo My favorite color is: ${FAVORITE_COLOR}!' > favorite_color.sh
# then, nano to add #!/bin/bash
```

In [ ]:
```
chmod +x favorite_color.sh
```

In [ ]:
```
PATH=${PATH}:~/shell-course/my_scripts
```

In [ ]:
```
export FAVORITE_COLOR=yellow
```

In [ ]:
```
cd ~
favourite_colour.sh
```

Summary

- there are two types of variables: "shell variables" and "environment variables"
    - only environment variables get passed to programs you call from the shell
    - you can turn a shell variable into an environment variable with `export`
- to retrieve the value of a variable, we need the `$` character (e.g. `$VAR` vs `VAR`)
- the shell will look for programs in your command in directories defined in the `$PATH` variable
- `$PATH` and other environment variables are set by startup files at the system and user level
- you can edit the startup files for your user in your home directory (e.g. `~/.bashrc`)

Final tips

The shell (bash) will be useful for you for:

- automating repetitive tasks
- keeping records of executed commands (through scripts) and re-using them
- access to remote computers like Compute Canada
- access to and understanding of tools in the neuroimaging world (many of the ones you will learn about this week)

With some experience, you'll probably find yourself often opening a terminal (running a shell) for something you could also do with your mouse or a graphical program.

Also consider:

- `bash` and other shells are great for many tasks, particularly when they involve changes to your files and directories
- But `bash` is not the right tool to create complex pipelines and programs like the ones needed for research analyses
- For these tasks, modern programming languages like `python` offer better error handling, control flow, debugging, and other features

What else can I do?

Check out the documentation for some other useful commands:

- **rsync**: local and remote file transfer (synchronization) that can detect and transfer differences only
- **tmux** (see also **this beginner's guide**): manage multiple terminal "windows" and keep sessions running in the background
- `cat -e` and **dos2unix**: check and convert line endings between Windows/Mac/Linux file formats (very useful if you work on a remote server that is a different OS)
- `sed` and `awk` **tutorial**: more advanced string manipulation/replacement and selecting specific sections of text
    - often used in a context with `grep`

Questions?

References

There are lots of excellent resources online for learning more about bash:

- The GNU Manual is *the* reference for all bash commands:
  http://www.gnu.org/manual/manual.html
- "Learning the Bash Shell" book:
  http://shop.oreilly.com/product/9780596009656.do
- An interactive on-line bash shell course: **https://www.learnshell.org/**
- The reference page of the software carpentry course:
  https://swcarpentry.github.io/shell-novice/reference.html