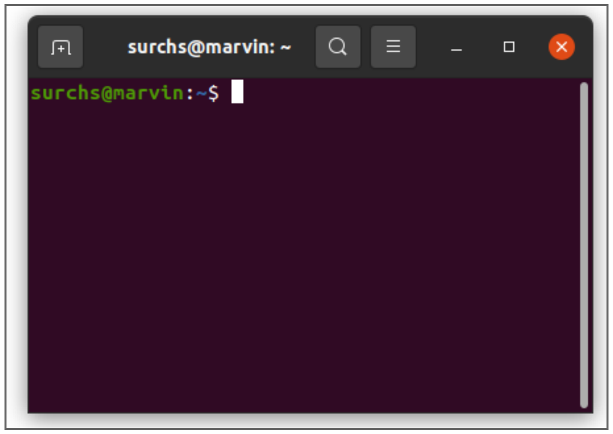
Introduction to the bash shell

A brief recap

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

Setting up for the exercises

1. Download the shell-course.zip materials from the course website: **https://neurodatascience.github.io/QLS612-Overview/lectures-materials.html**
2. Unzip the materials into your home directory (e.g. `/home/user-name` or `/Users/user-name`)
   Note: If you have already used these materials to follow the recorded lecture, please remove the old directory and make a fresh unzip
3. Open your terminal



```
surchs@marvin:~$
```

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

```
In [ ]:
```

```
echo $SHELL
```

In [ ]:

```
# /bin/bash
```

Bash command recap

In [ ]:

```
ls -l ~/shell-course
```

A shell command has 3 parts:

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

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` let's 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 names beginning with `.` are hiddden from listing by default

Refresher `cd`

`cd` -> changes your working directory

In [ ]:

```
pwd
```

In [ ]:

```
cd interesting_files
```

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/your-user-name`
- on Mac it is in `/Users/your-user-name`
- it contains your files and config files

In [ ]:

```
pwd
```

In [ ]:

```
cd
```

In [ ]:

```
pwd
```

In [ ]:

```
cd -
```

- `cd` without arguments brings you to your home direcory
- `~` is a shorthand for your home directory -> `cd ~` also brings you there

- your home directory has a path -> `cd /home/surchs` 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
- `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

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
- `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 extra `-r` flag

Recap: finding things

- we can print the structure of any given directory with `tree`
- `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 occurences 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
- `find` has a special `-exec` flag that let's you feed it's output to other bash programs

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:

- `man` -> opens the manual for a given command
- `whatis` -> gives a brief summary of a command
- `which` -> tells you where the program is located that is called by a command
- `--help` -> a flag that also shows the manual for many bash programs

In [ ]:

```
whatis wc
```

In [ ]:

```
which wc
```

Recap: variables and scripts

- the `$PATH` variable defines the directories where the shell will look for commands
- you can change `$PATH`, e.g. in your `~/.bashrc` file
- 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, e.g. `{` and `}` to differentiate it from other text
- 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}`)

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 [ ]:

```
ls -la
```

In [ ]:

```
tree
```

All these files are in `the_wrong_dir`, we want to move them to `the_right_dir`. Let's also use wildcards so we don't have to move each file separately.
Remember:

- `*` (the asterisk) will match any character 0 or more times. i.e. `*.txt` will match both `a.txt` and `any.txt` (any file ending in `.txt`)
- `?` (the questionmark) will match any character exactly once. i.e. `?.txt` will match only `a.txt` but not `any.txt`

Let's try this out with `ls` first.

In [ ]:

```
ls -lF
```

Exercise 1 Hints

- the `mv` command can move many files at once, as long as the `[new_path]` is a directory and not a file
- `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

Now 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]`.

Try for yourselves and I'll walk through in a moment.

In [ ]:

```
mv the_wrong_dir/my_file?.txt the_right_dir
```

In [ ]:

```
# Let's check that it has worked
tree
```

Exercise 1 - additional task

Now that everying in `the_right_dir` is good, we can get rid of `the_wrong_dir`. Remember:

- `rm` can remove files
- `rm` can only remove directories when the `-r` ("recursive") flag is set

In [ ]:

```
ls the_wrong_dir -la
```

In [ ]:

```
rm the_wrong_dir -r
```

Now `the_wrong_dir` is gone and there is no way to get it or it's contents back! Be **very careful** with `rm` commands, especially when you are 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 find out 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
```

Let's go into the `flying_circus` directory and print the length of each text file with `wc`

In [ ]:

```
ls -lF
```

In [ ]:

```
wc -l *.txt
```

Note: Instead of printing the output of `wc` to the screen, we can also redirect it. Here are some special characters that redirect the output that is normally printed to the screen (also called STDOUT):

- `|` 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_length.txt`
- `>>` redirects the output to a file and **appends** to this file if it exists. e.g. `wc -l *.txt >> file_length.txt`

Let's first write the output to a file with `>`. Let's call the output file `file_length.txt`.

Try for yourselves and I'll walk through in a moment.

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
```

This is not what we want. `sort` interpreted the numbers as text, but we want them interpreted as numbers. From `man sort`, we know that the `--numeric-sort` achieves this behaviour

In [ ]:

```
sort file_length.txt --numeric-sort
```

Lets' redirect this output as well, this time into a file called `sorted_length.txt`

In [ ]:

```
sort file_length.txt --numeric-sort > 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 sorted_length.txt -n 1
```

To find the shortest text file we have run:

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

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 above with the `|` character so the output of each command gets redirected to the next command rather than into a file.

Try for yourselves and I'll walk through in a moment.

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 (as in `python` the programming language). For example, the file `brian.txt` contains the script to `The life of Brian`. Let's say our goal is to make personalized copies of this file for the actor who play the role of `"Brian"` - with 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 with `"Brian"`. But we only want those strings that are denoting lines the character Brian says. We can do two things:

- search specifically for the string `"Brian:"` with the `:` character
- use the `^` character to only find occurences that are at the beginning of the line, i.e. `^Brian:`

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

In [ ]:

```
ls
```

In [ ]:

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

In [ ]:

```
head my_lines/Brians_lines.txt
```

That's nice. But how can we:

- easily create the lines for another role in this movie?
- remember 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

- easily change the role we create lines for?

Exercise 3 - let’s use scripts and variables

For this 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 quickly revisit the aspects of scripts and variables discussed in the recorded lecture!

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` 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
```

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

1. Let's start by re-running the command that created the lines for the role "Brian"
2. In the `~/shell-directory/flying_circus` directory, let's create an empty script file called `create_lines.sh`
3. Let's copy the `grep` command (or use the `history` of our shell) to the shell script
4. Let's 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` command: `chmod +x create_lines.sh`

In [ ]:

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

In [ ]:

```
history | tail -n 5
```

In [ ]:

```
# nano create_lines.sh
# OR
# history | tail -n 5 > create_lines.sh
# AND then: nano create_lines.sh
```

Try for yourselves and I'll walk through in a moment.

Hints:

- if you copy by hand, use the context menu (right-click). `CTRL+C` is reserved in `bash` to cancel commands
- in `nano`, remember to save (write) 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 means `CTRL+O`

Now let's

- give the script execution permission with `chmod +x`
- and see if this worked with `ls -lF`
- finally run our script with `./create_lines.sh`

Try for yourselves and I'll walk through in a moment.

In [ ]:

```
ls -lF
```

In [ ]:

```
chmod u+x create_lines.sh
```

In [ ]:

```
ls -lF
```

In [ ]:

```
./create_lines.sh
```

It works!

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

Recap variables

A 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

Let's look at this briefly

In [ ]:

```
MY_VAR=10
```

In [ ]:

```
echo $MY_VAR
```

In [ ]:

```
export MY_VAR
```

In [ ]:

```
printenv | grep MY_VAR
```

Our (next) step

1. Create a variable in our shell called `ROLE`
2. Assign the name of a role as the value of this variable (remember `=`)
3. Confirm the value was correctly assigned with `echo` (remember `$`)
4. Turn our variable into an environment variable so our script can see it (remember `export`)

THEN:

1. Edit our script `create_lines.sh` with `nano`
2. Replace the hard-coded string "Brian" with the variable `ROLE`

Try for yourselves and I'll walk through in a moment.

In [ ]:

```
ROLE=Brian
```

In [ ]:

```
echo ${ROLE}
```

In [ ]:

```
export ROLE
```

In [ ]:

```
# Confirm that "ROLE" is now an environment variable
printenv | grep ROLE
```

Finally replace the string of `"Brian"` in the script `create_lines.sh` with the value of the variable `${ROLE}`:
Remember:

- `$` to access the value of the variable
- variable names are case sensitive
- `{` and `}` are important when surrounding the variable with other text so bash can know where the variable name ends
- we can embed a variable in a string like this:

`"Hello World!"` -> `"Hello ${PLACE}!"`
notice how we can just put the variable inside of the string.
Try for yourselves and I'll walk through in a moment.

`grep "^Brian:" brian.txt > my_lines/brians_lines.txt`
`grep "^${ROLE}:" brian.txt > my_lines/${ROLE}s_lines.txt`

In [ ]:

```
cat create_lines.sh
```

In [ ]:

```
./create_lines.sh
```

Try changing the `ROLE` variable in the shell to other roles from the script and then run `create_lines.sh` again.
Here is a list with some other roles to try:

- Baby
- Balthasar
- Eremite
- Door
- Vendor

In [ ]:

```
ROLE=Door
```

In [ ]:

```
./create_lines.sh
```

Summary

- `grep` is a great tool to search within (text)files for occurences 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 have used environmental variables because we have seen them before. In practice there would have been 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 do too)

Exercise 4 - 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 recorded 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 [ ]:

```
# The -F flag for ls adds additional formatting for some directory contents
# The * is added to executable files
ls ../interesting_files -lF
```

In [ ]:

```
run_me.sh
```

So how can we add to the `PATH` variable?
Just like the `ROLE` variable in the previous exercise, we can re-assign the value of the `PATH` variable. We can simply add another directory to the list of directories in `$PATH` by using the `:` delimiter and then re-assigning the combined path to the `PATH` variable:
`PATH=${PATH}:/home/surchs/shell-course/interesting_files`
Try for yourselves and I'll walk through in a moment.

In [ ]:

```
echo $PATH
```

In [ ]:

```
PATH=${PATH}:/home/surchs/shell-course/interesting_files
```

Let's make sure this has worked:

In [ ]:

```
echo $PATH
```

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

In [ ]:

```
pwd
```

In [ ]:

```
run_me.sh
```

In [ ]:

```
run_me.sh
```

Summary

- 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 sytem and user level
- you can edit the startup files for your user in your home directory (e.g. `~/.bashrc`)
- to retrieve the value of a variable, we need the `$` character (e.g. `$VAR` vs VAR)
- 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`

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

- automating repetitive tasks
- keeping records of exectured 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

Questions?

Do you have any questions about what we just discussed or about bash in general?

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**