Summer Particle Astrophysics Workshop 2022: Introduction to Unix
Command Line Interface (CLI) and tools for scientific computing

David Gallacher
PhD Candidate - McGill University

david.gallacher@mail.mcgill.ca
May 3, 2022

About Me

- Completed Undergrad in Honours Experimental Physics from Carleton University (2019)
    - Bachelors Thesis project on measuring Cherenkov Radiation from DEAP-3600 Acrylic
- Completed Master's Degree in Particle Physics from Carleton University (2021)
    - Master's Thesis project was on building a small-scale R&D liquid argon detector using Silicon Photomulitpliers for readout (Argon-1) for ARGO
- PhD Candidate at McGill University working on nEXO photodetector R&D and the Light-only-liquid Xenon detector (LoLX)
    - Main research focus is on light readout from LXe detectors, signal analysis, and studying background rejection techniques for future LXe detectors
- Checkout my personal website for my full CV and information on research!
  www.davidgallacherphysics.com

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- **What is Linux?**
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library
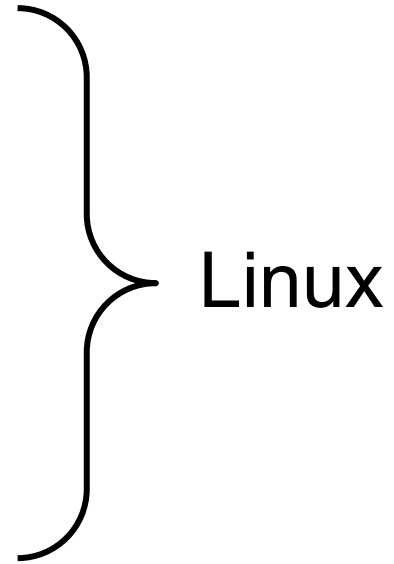
Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

What is Linux?

- Collection of Open Source Unix-like Operating Systems
    - Linux → Linux Is Not UNIX
- Unix Filesystem

    - Everything is a file (Everything)

    - Every file has a place. You don't have to put it there, but you should.

- Large library of software tools

- A shell scripting environment

    - Navigate the file system

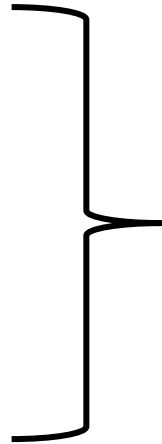   - Combine the software tools to accomplish complex tasks

Linux

What is Linux?

Linux Developer: Linus Torvalds

- Collection of open source unix-like operating systems
  - Linux → Linux Is Not UNIX

- Software Library

- Kernel

Linux Distribution (Ubuntu, Fedora, Debian, Arch,...)

- UNIX Filesystem

Why you should use Linux?

1. Its free and open source!
   a. Many flavors (distributions) to choose from, almost all of them are free.
   b. Ubuntu - Good all around option with long-term support options, CentOS/RHL for scientific computing, Lubuntu - Lightweight Ubuntu verison, great for bringing old PC's to life

2. When it works, it works!
   a. Can be left running for months/years without any issues.
   b. No need to reboot after installing software!
3. Runs on any hardware
   a. Can run full-scale desktop Linux options or ultra-lightweight linux for single-board-computers (SBCs) like Raspberry PI

The Most Useful Linux Troubleshooting Advice

First check "commandName –h" or --h or --help (Varies by program)

Next do "man commandName", if you can't find the information you need move on

Google:
Error Message/file + [Linux Distribution Name]

Check out the top 2-3 results, see what they agree on.

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- **Tutorial 1: Navigating the Filesystem**
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Tutorial 1: Navigating the Filesystem

1. Open Terminal (Mac) or Ctrl + Alt +T (Windows/Ubuntu)
   a. It may look like nothing, but it can do almost everything.
2. Check out your home folder

   ls
   (Lists the contents of your home folder)

   ls -a
   (Includes "hidden" files, i.e., files whose name starts with '.')

   (Files and Directories will probably be coloured differently)

Tutorial 1: Navigating the Filesystem

1. Move into a new directory (e.g.eieioo)
   cd eieioo

```
[dgallach@cedar1 ~]$ cd eieioo/
[dgallach@cedar1 eieioo]$ ls
[dgallach@cedar1 eieioo]$
```

Username

Machine name

Current folder

Tutorial 1: Navigating the Filesystem

1. Move into a new directory (e.g. eieioo)
   cd eieioo

```
[dgallach@cedar1 ~]$ cd eieioo/
[dgallach@cedar1 eieioo]$ ls
[dgallach@cedar1 eieioo]$
```

2. Is there anything there?
   ls
   (maybe not)
   ls -a
   (you will notice '.' and '..')
3. What does .. mean?
   cd ..
4. pwd (print working directory)

Linux Shorthand:

. → Current Directory

.. → Parent Director

~ → Home Folder (Also called $HOME)

Tutorial 1: Navigating the Filesystem

1. Try using an absolute file path
    cd /home/[username]/eieioo

2. You can use ~ as a shortcut for your home folder
    cd ~/eieioo

Let's start playing around with files:

1. Copy a file into your Documents folder (commands are equivalent)
    absolute paths: cp /home/[username]/.bashrc /home/[username]/Documents/
    relative paths: cp ../.bashrc ./

2. You can copy a file to a new filename
    cp ~/.bashrc ./dummyfile

Tutorial 1: Navigating the Filesystem

1. Make a new directory:
    mkdir dummydir

2. Move your dummyfile into dummydir:
    mv dummyfile dummydir

3. Check that is worked:
    ls dummyfile
    (should return ls: cannot access 'dummyfile': No such file or directory)

    ls dummyfolder/dummyfile
    (should not give an error)

Tutorial 1: Navigating the Filesystem

Let's make a sample Python file with ***nano (also check out vim and emacs)***.

1. Open your file (notice the .py ending): nano test.py

2. Add some lines

   print('Hello McDonald Institute!')

3. To save and exit:

   a. Ctrl+x - To exit

   b. Then type "y" to save changes ("n" to not save")

4. Run your python file: python test.py

Tutorial 1: Navigating the Filesystem

Let's clean up after ourselves

1. rm ./.bashrc
   (deletes the .bashrc file we copied into Documents)

2. rm -r dummydir
   (recursively deletes anything inside dummydir and then deletes dummydir)

Tutorial 1: Navigating the Filesystem

Summary of Commands:

1. ls lists contents of current directory
2. ls /path/to/dir lists contents of a specific directory
3. cd /path/to/dir changes current directory
4. cp copies files
5. mv moves/renames files/directories
6. mkdir makes directory
7. rm deletes files/directories

Specific options for all of these commands can be found using the --help flag

More detailed instructions can be found using man (e.g., try man ls or man echo)

Activity 1: Navigating the Filesystem

1. Make two directories: *testing* and *EIEIOO_Scripts*

2. Make two sample python scripts in the *testing* directory

3. Move one script to *EIEIOO_Scripts*

4. Copy the other script to *EIEIOO_Scripts*

5. Delete the *testing* folder

6. Run your python scripts

```
physuser@b77ad2eaee59:~/EIEIOO_Scripts$ ls
test.py   test2.py
```

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- **Tutorial 2: File Permissions**
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Tutorial 2: File Permissions

Lets have a look at the root folder again:

ls -l /

```
lrwxrwxrwx      1 root root       8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x  35613 root root 1388544 May   1 12:19 scratch
drwxr-xr-x      2 root root       6 Apr 10  2018 srv
dr-xr-xr-x     13 root root       0 Jan 15 23:08 sys
drwxrwxrwt   1384 root root  140980 May   1 12:27 tmp
drwxr-xr-x     13 root root     155 Oct 15  2018 usr
drwxr-xr-x     19 root root     265 Oct 15  2018 var
drwxr-xr-x      3 root root      21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```
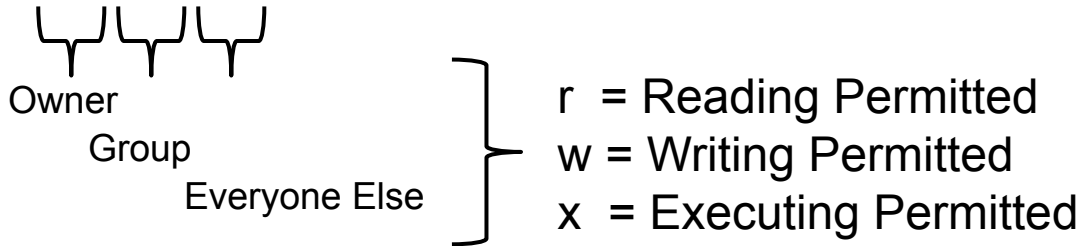
What is all this stuff?

Tutorial 2: The Linux Filesystem

```
lrwxrwxrwx       1 root root       8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x   35613 root root 1388544 May   1 12:19 scratch
drwxr-xr-x       2 root root       6 Apr 10  2018 srv
dr-xr-xr-x      13 root root       0 Jan 15 23:08 sys
drwxrwxrwt    1384 root root  140980 May   1 12:27 tmp
drwxr-xr-x      13 root root     155 Oct 15  2018 usr
drwxr-xr-x      19 root root     265 Oct 15  2018 var
drwxr-xr-x       3 root root      21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

The first character tells you what type of file: - (regular file), d (directory), l (link)

Tutorial 2: File Permissions

The next nine characters gives you three different sets of "permissions" for the file
- ○ Three different levels of control over the file

```
drwxrwxrwt    1384 root root   140980 May  1 12:27 tmp
drwxr-xr-x      13 root root      155 Oct 15  2018 usr
drwxr-xr-x      19 root root      265 Oct 15  2018 var
drwxr-xr-x       3 root root       21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

Owner
    Group
        Everyone Else

r = Reading Permitted
w = Writing Permitted
x = Executing Permitted

Tutorial 2: File Permissions

```
lrwxrwxrwx      1 root root        8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x  35613 root root  1388544 May  1 12:19 scratch
drwxr-xr-x      2 root root        6 Apr 10  2018 srv
dr-xr-xr-x     13 root root        0 Jan 15 23:08 sys
drwxrwxrwt   1384 root root   140980 May  1 12:27 tmp
drwxr-xr-x     13 root root      155 Oct 15  2018 usr
drwxr-xr-x     19 root root      265 Oct 15  2018 var
drwxr-xr-x      3 root root       21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

Number of links/directories inside a link/directory

Tutorial 2: File Permissions

```
lrwxrwxrwx       1 root root        8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x   35613 root root 1388544 May  1 12:19 scratch
drwxr-xr-x       2 root root        6 Apr 10  2018 srv
dr-xr-xr-x      13 root root        0 Jan 15 23:08 sys
drwxrwxrwt    1384 root root  140980 May  1 12:27 tmp
drwxr-xr-x      13 root root      155 Oct 15  2018 usr
drwxr-xr-x      19 root root      265 Oct 15  2018 var
drwxr-xr-x       3 root root       21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

Owner of the file

You can modify who owns a file with: chown [username] [filename]

Tutorial 2: File Permissions

```
lrwxrwxrwx      1 root root       8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x  35613 root root 1388544 May  1 12:19 scratch
drwxr-xr-x      2 root root       6 Apr 10  2018 srv
dr-xr-xr-x     13 root root       0 Jan 15 23:08 sys
drwxrwxrwt   1384 root root  140980 May  1 12:27 tmp
drwxr-xr-x     13 root root     155 Oct 15  2018 usr
drwxr-xr-x     19 root root     265 Oct 15  2018 var
drwxr-xr-x      3 root root      21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

Group that owns the file

You can add users to a group using: usermod -aG [groupname] [username] (or chgrp -options GROUP FILE)

Tutorial 2: File Permissions

```
lrwxrwxrwx       1 root root         8 Oct 15   2018 sbin -> usr/sbin
drwxr-xr-x   35613 root root   1388544 May   1 12:19 scratch
drwxr-xr-x       2 root root         6 Apr 10   2018 srv
dr-xr-xr-x      13 root root         0 Jan 15 23:08 sys
drwxrwxrwt    1384 root root    140980 May   1 12:27 tmp
drwxr-xr-x      13 root root       155 Oct 15   2018 usr
drwxr-xr-x      19 root root       265 Oct 15   2018 var
drwxr-xr-x       3 root root        21 Jan 30   2019 wlcg
[dgallach@cedar1 /]$
```

Size of the file in bytes. Try using ls -lh for human-0readable sizes.

Tutorial 2: File Permissions

```
lrwxrwxrwx        1 root root        8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x    35613 root root  1388544 May   1 12:19 scratch
drwxr-xr-x        2 root root        6 Apr 10  2018 srv
dr-xr-xr-x       13 root root        0 Jan 15 23:08 sys
drwxrwxrwt     1384 root root   140980 May   1 12:27 tmp
drwxr-xr-x       13 root root      155 Oct 15  2018 usr
drwxr-xr-x       19 root root      265 Oct 15  2018 var
drwxr-xr-x        3 root root       21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

Date of last modification

Tutorial 2: File Permissions

```
lrwxrwxrwx        1 root root          8 Oct 15  2018 sbin -> usr/sbin
drwxr-xr-x  35613 root root    1388544 May   1 12:19 scratch
drwxr-xr-x       2 root root          6 Apr 10  2018 srv
dr-xr-xr-x      13 root root          0 Jan 15 23:08 sys
drwxrwxrwt    1384 root root     140980 May   1 12:27 tmp
drwxr-xr-x      13 root root        155 Oct 15  2018 usr
drwxr-xr-x      19 root root        265 Oct 15  2018 var
drwxr-xr-x       3 root root         21 Jan 30  2019 wlcg
[dgallach@cedar1 /]$
```

Filename

Tutorial 2: File Permissions

Ok so we have seen how to give ourselves ownership (chown) or group membership (usermod -aG).

What about everyone else? Can we modify the owner/group permissions?

chmod u=rwx,g=rx,o=r [filename]

u = user = owner

g = group

o = other = everyone else

Everyone can read, write, and execute: chmod 777 <filename>

Tutorial 2: File Permissions

Ok so we have seen how to give ourselves ownership (chown) or group membership (usermod -aG).

What about everyone else? Can we modify the owner/group permissions?

    chmod u=rwx,g=rx,o=r [filename]

u = user = owner

g = group

o = other = everyone else

Different Permutations:

0 – no permission

1 – execute

2 – write

3 – write and execute

4 – read

5 – read and execute

6 – read and write

7 – read, write, and execute

Tutorial 2: File Permissions

The dangerous but useful: **chmod 777 <filename>**

Everyone can read, write, and execute this file.

Exercise 2: File Permissions

1. Change one of your sample Python files permissions so that you get the following error when you try to run it:

    python: can't open file 'test.py': [Errno 13] Permission denied

2. Figure out how to successfully run it again

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- **Tutorial 3: The Builtin Software Library**

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Tutorial 3: The Builtin Software Library

1. Let's find where python is:
   whereis python
   For me, there are multiple python versions installed.

2. cd /usr/lib/python3

   Let's check out the Python versions installed here.

Tutorial 3: The Builtin Software Library

1. Lets see how many python files there are
    find . -name '*.py'

    Look 'here' for files whose name matches the pattern *.py (*=wildcard)
    (Ok, that's a lot of files!)

2. grep -r --include '*.py' '= list()'

    Recurse down the filesystem, looking inside files that end with .py searching for '=
    list()'

Tutorial 3: The Builtin Software Library

1. Now to count them up
    grep -r --include '*.py' '= list()' | wc -l

    The character | "pipes" the output of grep into wc which counts the number of
    words it is (or in this case lines due to -l)

2. Ok now lets try to repeat that with my_list = []
    grep -r --include '*.py' '= []' | wc -l

    You will get an error:
    The '[]' characters are special and must be "escaped"
    grep -r --include '*.py' '= \[\]' | wc -l

Tutorial 3: The Builtin Software Library

Great!

So how many instances did you find?

More generally, grep is extremely useful for finding strings (or a RegExp) in text files when you can't remember which file it's in.

Also, very useful - piping outputs into grep.

For example:

Ifconfig   - Returns a long list of all your network interfaces. ('ip a' instead on some linux systems)

Ifconfig | grep "inet" - "|" is for "piping" the output of the left-hand side, as input into the right-hand side (Read more here) , "inet" is your IP address

Tutorial 3: The Builtin Software Library

Some commands I commonly use:

find: find files

grep: search for strings/patterns inside text files

top or htop (fancy version): similar to task manager

df -h: check how much free disk space is available on your system

du -h : Check how much space is used in the current directory

Tip: Google "How to do <blank> command line linux"

Exercise 3: The Builtin Software Library

Determine how many times 'import' is used in your Python3 installation. For me, it's 18!

Lastly, what about a job I need to quit?

Ok now we started a rogue process which will never finish and will eat up our hard drive. What do we do?

Use the pid to kill the process!

kill 25879

If you do not remember the pid, use jobs -l to find the jobs currently running in a given shell.

Check for tasks+PIDs running on your computer with ps -aux , to search for a specific task, pipe to grep! ps -aux | grep "taskname"

```
n$ jobs -l
[1]+ 25879 Running                      while sleep 1; do
    date >> curr_time;
done &
```

Shell Scripting

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- **What is a shell?**
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

What is the shell?

We've been working in the 'terminal' or 'command-line interface' these are examples of shell programs.

Shell programs take keyboard inputs and turn them into instructions for the computer

Examples include: Mac Terminal, Windows PowerShell, Linux Terminals

Different Shell types are around: bash (Bourne again shell), csh (C-Shell), zsh (Z-shell, default on Mac)

He doesn't know how to use the three seashells!

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- **Tutorial 1: Your shell profile**
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Tutorial 1: Shell Profiles

You may have noticed that my shell has colour highlighting and other quality of life features

Your shell has default options that you can configure.
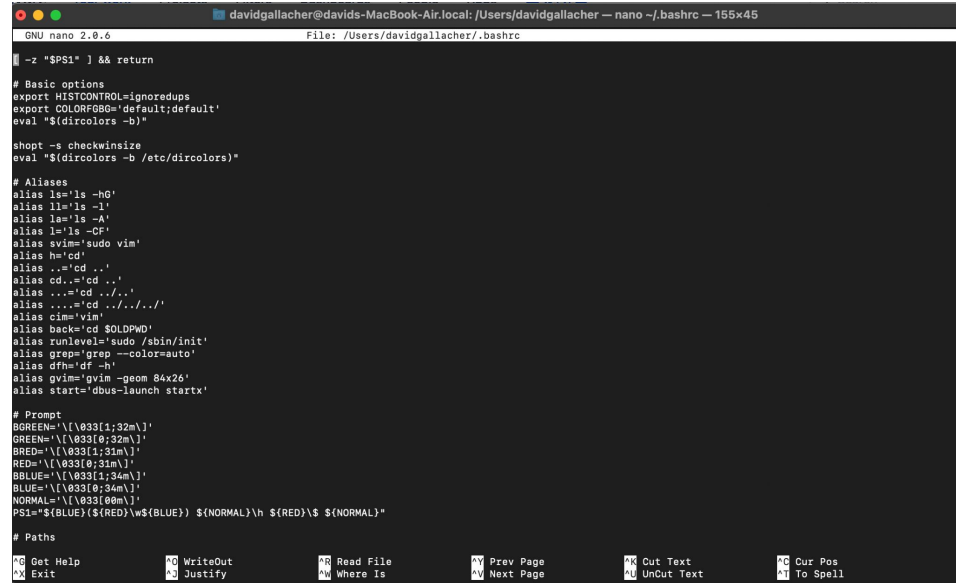
Take a look at your shell profile~

> Nano ~/.bashrc
>
> nano ~/.bash_profile
>
> Or on Mac
>
> nano ~/.zprofile or nano ~/.zshrc

Tutorial 1: Shell Profiles

Interactive shells (What happens when you open the terminal) call ~/.bashrc before starting

This is a useful place to set any variables we may want to use in CLI, as well as calling any scripts we may want to run before starting a CLI session

Let's do an example, open your terminal and call:

    nano ~/.bashrc

We will add a new variable and a print statement.

Add the following lines:

    MYVAR="Hello McDonald Institute!" - this sets a variable within the bashrc script, which can be called later on

    export MYVAR - Tells the shell to remember this variable, now we can call it from the CLI, this is now an "environment variable"

    echo $MYVAR - "echo" prints the argument to the command line, this will print the contents of MYVAR

Exit and save (Ctrl+x then 'y')

Tutorial 1: Shell Profiles

Now we need to re-load the ~/.bashrc script, we can either do that by opening a new terminal or by resourcing the script

    source ~/.bashrc

We should see our print statement with the variable contents
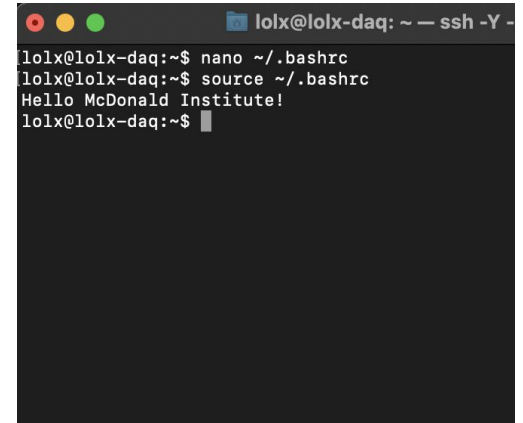
Next let's print the statement from the terminal to check if it was exported properly

    echo $MYVAR - Print the contents of this environment variable!

Lastly we can assign "aliases" to commands, so that we can invoke them later-on. Very handy for long commands you don't want to type out!

See here for details on making aliases.

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- **Tutorial 2: Using shell scripts**
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Tutorial 2: Shell Scripts

We just saw our first example of a shell script, a set of command-line instructions we can call from a script instead of manually

This opens lots of doors for us!

Shell scripts typically end in ".sh" can can be called like:

      bash myscript.sh

Or   sh myscript.sh

Or   ./myscript.sh - This option requires some additional steps!

Tutorial 2: Shell Scripts

Let's make a shell script and test it out

From the terminal do:

nano myscript.sh

Let's add the following lines

#This script creates a variable and prints it! - Comments in Shell scripts are prepended with '#'

var=100 - Make a variable called var, and set it to 100 (Careful here, 'var = 100' returns an error!)

echo $newvar

Save the script and run it with bash myscript.sh

Tutorial 2: Shell Scripts

Now we will expand the test script to do more things

We can call programs and other commands from the shell script.

Anything you can do in the command line interface, you can do in a shell script!

Tutorial 2: Shell Scripts

We can pass arguments to our scripts as well

Bash supports up to 9 arguments directly from the command line

Let's modify our script, add the following lines:
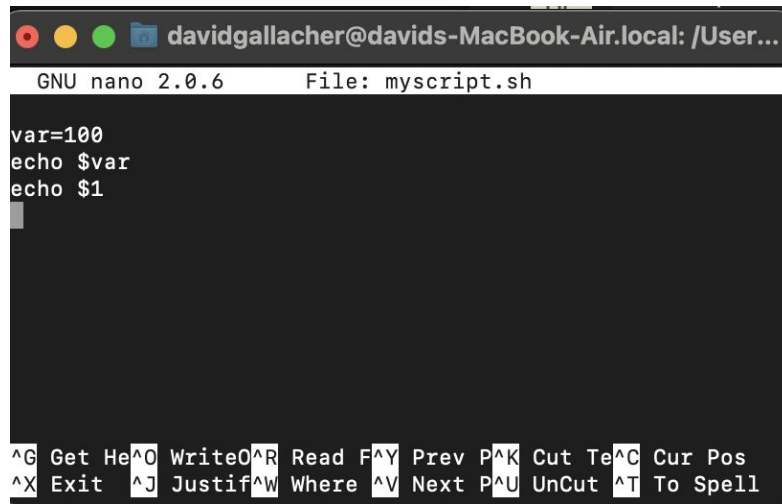
echo $1 - This prints the first command-line argument, similarly, $0-$9 print any other arguments.

$0 is reserved for the script name (myscript.sh)

Save and exit, and then call the script, this time passing an argument:

bash myscript.sh "Hello McDonald Institute"

```
GNU nano 2.0.6        File: myscript.sh

var=100
echo $var
echo $1




^G Get He^O WriteO^R Read F^Y Prev P^K Cut Te^C Cur Pos
^X Exit  ^J Justif^W Where ^V Next P^U UnCut ^T To Spell
```

Our script so far!

```
(base) (~) davids-MacBook-Air $ nano myscript.sh
[(base) (~) davids-MacBook-Air $ bash myscript.sh "Hello McDonald Institute"
100
Hello McDonald Institute
(base) (~) davids-MacBook-Air $
```

Tutorial 2: Shell Scripts

Lastly, we will call python from our shell script and save the output to a variable that gets printed in a formatted message

Add the following lines to your script:

pythonVersion=$(python –version)

printf "The python version currently installed is : %s \n" "$pythonVerison"

Then we can save the file and call it again, this time we call it directly:

./myscript.sh "Hello McDonald Institute"

Tutorial 2: Shell Scripts

… Then we can save the file and call it again, this time we call it directly:

> ./myscript.sh "Hello McDonald Institute"

This will return an error! We need to set the file permissions to make this script "executable", and we need to tell the shell what scripting language to use.

First, add this line to the very start of your script:

> #!/usr/bin/env bash -This is called "shebang" and tells the computer which interpreter to use to understand the instructions in the rest of the file, here we're using bash

Now we need to make it executable:

> chmod u+=x myscript.sh

Now we should be able to call it directly!

> ./myscript.sh

Tutorial 2: Shell Scripts

Lots of options for automating tasks in shell scripts:

Some examples that I've used in physics before:

- Script that analyzes data using a python script then sends the output to another script
- Create a for-loop in the bash script to process many data files and rename their outputs and sync to cloud
- Script that runs continuously in the background and looks for new data files being written, then analyzes them automatically
- Much more..

Lots of power in shell scripts for making your life easier!

Google is your best friend, if you're thinking of automating a task try searching "do `task` in bash script"

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- **Tutorial 3: ssh**

Part 3: Containers (Time Permitting)

- What is a container?
- Tutorial 1: Using a container

Tutorial 3: SSH - The secure shell

SSH or "Secure Shell" is a network communication protocol. It allows you to obtain a virtual terminal or "shell" from a remote machine.

Using SSH you can connect to a machine with an SSH-server and run command-line-interface instructions over the network

Widely used in physics for connecting to computing clusters, remote workstations, experimental equipment etc..

SSH is an encrypted network protocol, all traffic is encrypted and secure!

Tutorial 3: SSH - The secure shell

SSH works in two parts, a client and a server

A client runs on your PC and communicates with an SSH server running on the remote machine over the network

This can be done directly in a Mac/Linux Terminal, on Windows in Powershell or by using a GUI for SSH like Putty

Example:

ssh -Y username@remote-address.com

Remote addresses can be IP addresses or domain names!
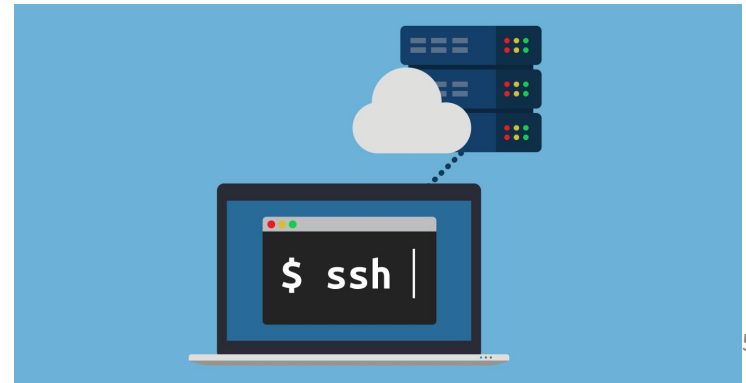
Tutorial 3: SSH - The secure shell

SSH requires you to log in to the target machine, this can be done in 2 ways

1. Password authentication
   a. You enter the password for the user you are trying to obtain a shell from
   b. Susceptible to attacks! Password authentication left open on a public IP can be a serious security flaw
2. Key-exchange
   a. You copy your "public" key to the server, which it uses to verify connections later
   b. For an in-depth overview of key-exchange see this great video!
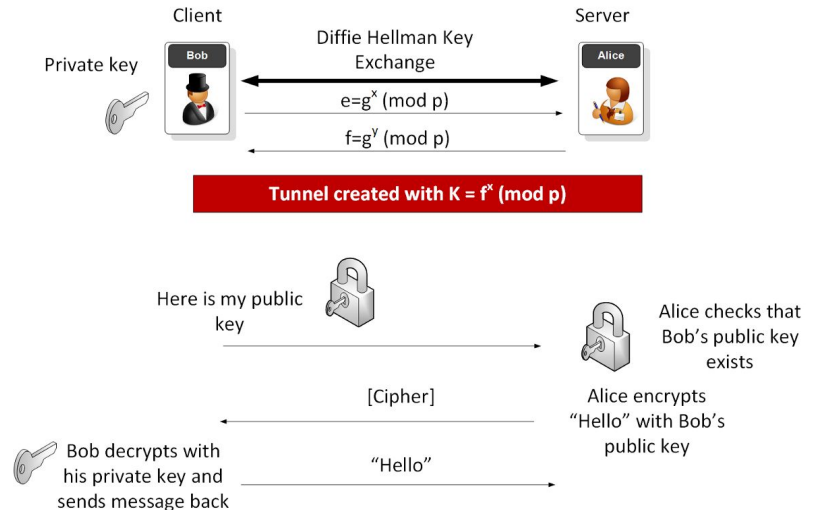
Tutorial 3:

Now we will make an ssh public key, copy it to a remote server and establish a connection

Let's first check if a key exists by running in the terminal:

    ls -la ~/.ssh/

```
            id_rsa        known_hosts  test-key
(base) (~) davids-Air $ ls -la ~/.ssh/
total 64
drwxr-xr-x   9 davidgallacher  staff   288B 17 Mar 14:11 .
drwxr-xr-x+ 74 davidgallacher  staff   2.3K 29 Apr 16:09 ..
-rw-r--r--   1 davidgallacher  staff   416B 17 Sep  2021 config
-rw-------   1 davidgallacher  staff   2.6K 20 Jul  2021 id_rsa
-rw-r--r--   1 davidgallacher  staff   593B 20 Jul  2021 id_rsa.pub
-rw-r--r--   1 davidgallacher  staff   9.2K 25 Mar 16:51 known_hosts
drwx------   2 davidgallacher  staff    64B 28 Apr 18:11 socket
-rw-------   1 davidgallacher  staff   3.4K 16 Sep  2021 test-key
-rw-------   1 davidgallacher  staff   754B 13 Sep  2021 test-key.pub
(base) (~) davids-Air $ 
```

I already have two keys, "id_rsa" (private and public) and "test-key"

Tutorial 3: SSH - The secure shell

Let's make a new key:

ssh-keygen -t rsa -b 4096 -f "~/.ssh/my_new_key"

Encryption type

Size of key

Filename
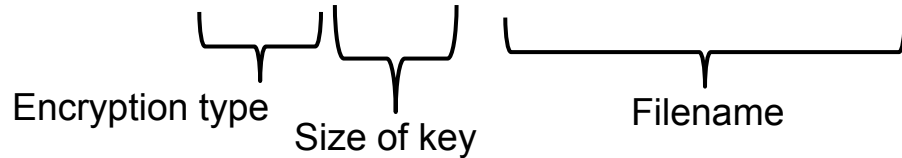
This will prompt you for a password for your key, you can leave it blank (Hit enter) or make a password. Some servers require you to have password protected ssh keys!

You will see a random-art image representing your new key

Tutorial 3: SSH - The secure shell

Now that we have created an ssh key, we can connect to a remote server without password authentication

But first the server needs to know our key!

Two ways of doing this:

1. For some systems that have password authentication + key exchange options
   - ssh-copyid -i "~/.ssh/my_new_key.pub" user@remote-address - copies our public key to the server to store, need passwords!
2. Other systems will require you to send your plaintext public key (Contents of ~/.ssh/my_new_key.pub) to the system administrators to store manually. No password access to the server.

Lots of power to simplify things, including using the SSH config file, see here for more information on setting up your own configurations.

Tutorial 3: SSH - The secure shell

What if we need to copy files that we made on a remote machine to a local machine?

Here are two options:

'scp' - 'Secure copy', copy data from a remote location to local, or vice-versa, is encrypted by default
- Can specify address and port for transfer
- Example to copy from remote to a local directory, through port '2205' on remote host
    - scp -P 2205 username@remote_host:/path/to/file/to/copy /path/to/local/directory/to/store
- Example to copy a local file to a remote directory, through port '2205' on remote host
    - scp -P 2205 /path/to/local/file/to/copy/file.txt username@remote_host:/path/to/remote/directory/to/store

'rsync' - "Remote Sync", tool for copying files, or synchronizing a remote directory to a local one
- Faster than scp, but not encrypted by default (Can encrypt with an ssh tunnel)
- Example to sync two directories between remote and local
    - rsync -artzP username@remote_host:/home/username/dir1 /place/to/sync/on/local/machine

Google and GNU/Linux manual pages are your best friends when using command-line tools!

Tutorial 3: SSH - The secure shell

Lots of doors open with ssh connections

- Running remote commands on other PCs
- Connecting to remote instruments to control (Raspberry Pi's and other SBC's are ideal for this!)
- Connect to computing clusters to perform large scale computing tasks (Such as the Digital Research Alliance clusters, formerly Compute Canada)
- Access a computer on your home network from the office, or vice-versa, to work remotely

Containers

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- **What is a container?**
- Tutorial 1: Using a container

What is a container?

Containers are software development tools that are growing in usage in scientific research

Containers package together the application with the required libraries and dependencies to run on any operating system or hardware

Similar to virtual machines (VMs) but differ in important ways
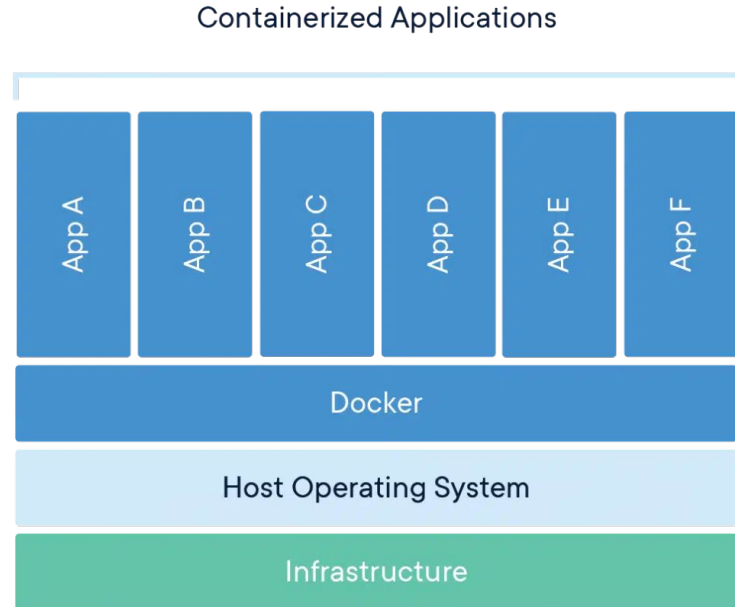
Image from Docker (link)

Container vs VM

VM's are instances of operating systems that run separate from the main OS on the hardware

For example, you can run a Windows VM on a Mac in order to run Windows specific software

VM's are large (Several GB's) and contain everything required for an operating system to work

Containers are lightweight (MB's) and contain only what's required for a specific application to run.

Containers run like applications on the host OS and virtualize resources
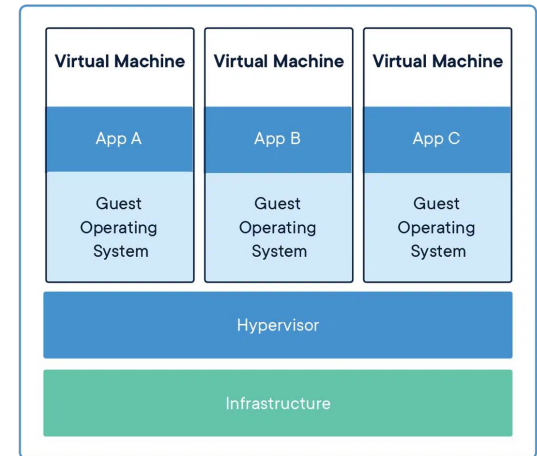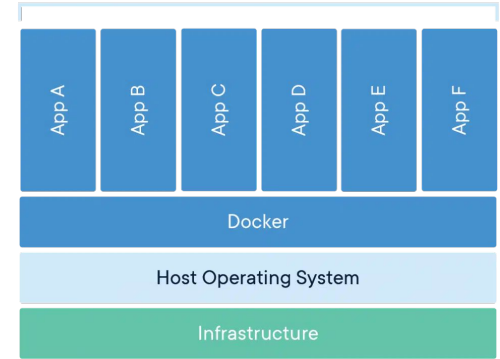
Images from Docker (link)

68

Containers for Scientific Computing

Containers have a lot of value in scientific computing, here are some common considerations for scientific software

- Research often uses custom software packages where scientists are developers
- Creating cross-platform reliable software with idiosyncratic differences from different compiler versions, hardware and OS's is a challenge
- Usability is a major consideration for scientific software, we need tools that will work reliably with minimal upkeep, so we can spend our time doing science instead of solving software related bugs
- Reproducibility, Some simulation software (GEANT4 or ROOT for example) will have different implementations of **physics,** depending on what version you use! So we need to specify exact versions for use in order to reproduce our results

Containers can address all of these points!

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Shell Scripting

- What is a shell?
- Tutorial 1: Your shell profile
- Tutorial 2: Using shell scripts
- Tutorial 3: ssh

Part 3: Containers (Time Permitting)

- **What is a container?**
- Tutorial 1: Using a container

Tutorial 1: Using a container

The most popular container engine is Docker

We will use this for the tutorial, you can download docker here: https://www.docker.com/get-started/

We will follow the sample tutorial here (link)

This will also allow us to test out our command-line interface techniques

Tutorial 1: Using a container

To run our first container, we simply type:

docker run hello-world

```
(base) (~) davids-Air $ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
7050e35b49f5: Pull complete
Digest: sha256:10d7d58d5ebd2a652f4d93fdd86da8f265f5318c6a73cc5b6a9798ff6d2b2e67
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Tutorial 1: Using a container

To run our first container, we simply type:

docker run hello-world

```
(base) (~) davids-Air $ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
7050e35b49f5: Pull complete
Digest: sha256:10d7d58d5ebd2a652f4d93fdd86da8f265f5318c6a73cc5b6a9798ff6d2b2e67
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Since we didn't have "hello-world" image installed, docker downloaded it for us before running as a container

Tutorial 1: Using a container

Now we will install another container from Docker's repository of containers (See here for more containers)

docker pull alpine

Then we can see all the images we've installed from docker

docker images

```
(base) (~) davids-Air $ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
9981e73032c8: Pull complete
Digest: sha256:4edbd2beb5f78b1014028f4fbb99f3237d9561100b6881aabbf5acce2c4f9454
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
(base) (~) davids-Air $ docker images
REPOSITORY         TAG        IMAGE ID       CREATED         SIZE
alpine             latest     3fb3c9af89a9   4 weeks ago     5.32MB
hello-world        latest     46331d942d63   6 weeks ago     9.14kB
docker101tutorial  latest     6ad88b1adc44   5 months ago    27.2MB
alpine/git         latest     4ee6a3b79e0c   5 months ago    27.1MB
(base) (~) davids-Air $
```

Tutorial 1: Using a container

Now we will install another container from Docker's repository of containers (See here for more containers)

docker pull alpine

Then we can see all the images we've installed from docker

docker images

```
(base) (~) davids-Air $ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
9981e73032c8: Pull complete
Digest: sha256:4edbd2beb5f78b1014028f4fbb99f3237d9561100b6881aabbf5acce2c4f9454
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
(base) (~) davids-Air $ docker images
REPOSITORY         TAG       IMAGE ID        CREATED         SIZE
alpine             latest    3fb3c9af89a9    4 weeks ago     5.32MB
hello-world        latest    46331d942d63    6 weeks ago     9.14kB
docker101tutorial  latest    6ad88b1adc44    5 months ago    27.2MB
alpine/git         latest    4ee6a3b79e0c    5 months ago    27.1MB
(base) (~) davids-Air $
```

Tutorial 1: Using a container

Alpine is a lightweight linux distro, available as a docker container

 We can use alpine + docker to run linux command-line instructions

docker run alpine ls -l

When we call "run" we're telling docker to create the container for the "alpine" image that we pulled earlier, and issue the command 'ls -l'

Tutorial 1: Using a container

We can also use and "interactive" flag to keep the container running after issuing the command, try:

docker run -it alpine /bin/sh

Then run:

echo "Hello McDonald Institute"

We can exit with "exit" or CRTL+C

Tutorial 1: Using a container

Lastly, let's check our tasks to see if any containers are running:

docker ps

You should see an empty line, since none of the containers we started were running as "Daemons", which tells the container to run continuously in the background

To see what containers we've run recently, we can append an -a or "all"

docker ps -a

```
[(base) (~/mcdonald_institute_summer_workshop_2022) davids-Air $ docker ps -a
CONTAINER ID   IMAGE         COMMAND                CREATED          STATUS                    PORTS     NAMES
489de7610038   alpine        "/bin/sh"              9 minutes ago    Exited (0) 9 minutes ago            infallible_yalow
a762a6ef467f   alpine        "echo 'Hello McDonal…" 13 minutes ago   Exited (0) 13 minutes ago           inspiring_curie
fa44a217009c   alpine        "ls -l"                17 minutes ago   Exited (0) 17 minutes ago           blissful_archimedes
ced271379bc6   hello-world   "/hello"               41 hours ago     Exited (0) 41 hours ago             nervous_elgamal
```

Tutorial 1: Using a container

We've used some phrases without explaining in detail, here are some definitions (Copied from [here](#))

- *Images* - The file system and configuration of our application, which are used to create containers. To find out more about a Docker image, run docker inspect alpine. In the demo above, you used the docker pull command to download the **alpine** image. When you executed the command docker run hello-world, it also did a docker pull behind the scenes to download the **hello-world** image

- *Containers* - Running instances of Docker images — containers run the actual applications. A container includes an application and all of its dependencies. It shares the kernel with other containers, and runs as an isolated process in user space on the host OS. You created a container using docker run which you did using the alpine image that you downloaded. A list of running containers can be seen using the docker ps command.

- *Docker daemon* - The background service running on the host that manages building, running and distributing Docker containers.

- *Docker client* - The command line tool that allows the user to interact with the Docker daemon.

- *Docker Store* - A [registry](#) of Docker images, where you can find trusted and enterprise ready containers, plugins, and Docker editions.

Containers Closing Remarks

Containers are very useful tools for running specific applications/custom software on arbitrary systems

Loads of applications (Especially web-apps) available as containers

Scientific software (Custom simulation packages, version dependent effects etc..) are fertile ground for containers

Some further steps:

- Make your own image to learn more about Docker (link)
- Look for an image of a software you're considering using

Workshop Closing Remarks

Command-line interface tools are powerful and plentiful!

Making use of existing tools, and doing some research on what's possible before starting a project or a new task can make your life much easier! Less time spent on software wrangling = More time for science

Google and online manuals are your best friends for learning, and talk to your colleagues and collaborators about the tools they like best!

A word of caution, be very careful copying and pasting commands from online forums, malicious actors exist and a seemingly benign looking set of commands can become a serious security issue!

Check multiple sources before running commands from the internet! If you're unsure, reach out to colleagues for help.

Always think before you 'sudo'
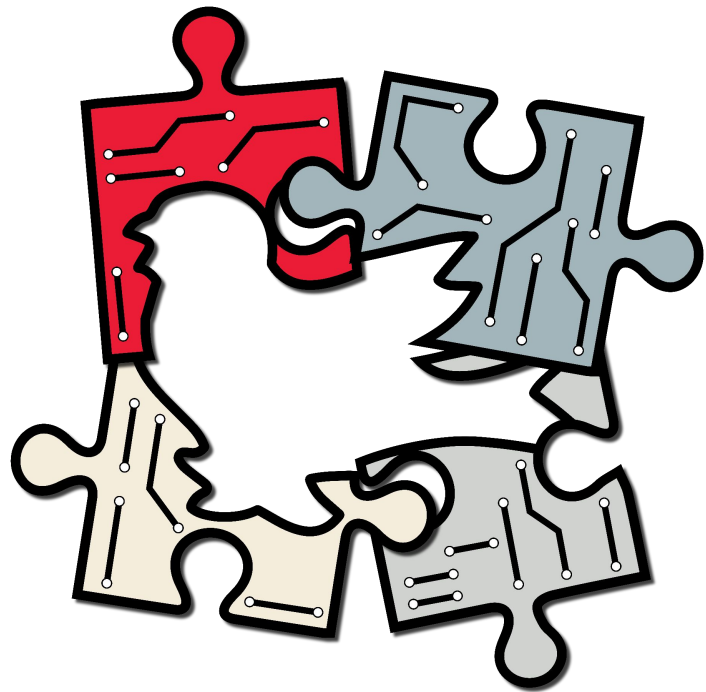
See you in the fall at the McGill Physics Hackathon?

Questions?

You can also email me after at david.gallacher@mail.mcgill.ca

Resources

- [How to install git on any OS](#)
- [A nice ELI5 git series](#)
- ["What is git" from Atlassian](#)
- [Basic git tutorial](#)
- [Reference for adding local git projects to the cloud](#)
- [An in-depth summary of remote branches](#)
- [Tutorial on how to deal with merge conflicts](#)
- [SSH Beginner Guide](#)
- [Piping in Linux](#)
- [Docker Getting Started](#)

Backup

Tutorial extra: Networked Computing

Open a secure connection to a shell instance on a remote machine:

ssh [username]@[remote machine]

e.g., lmacka3@mimi.cs.mcgill.ca   ([register here](#))

[Do what you need to do on the remote machine]

Close the shell:

exit

Tutorial extra: Networked Computing - Copying Files

You may need some of your files on remote machine. Use scp [source] [target]!

Lets make a dummy file: touch ~/foo

Copy the file to a remote machine:

scp ~/foo lmacka3@mimi.cs.mcgill.ca:/home/cnd/lmacka3

You may need to use ssh to figure out the remote path you are copying to.

You can also use rsync to transfer only new files

Demonstration extra: Networked Computing - Copying Files

I will now share my terminal to demonstrate copying some files on Compute Canada's Beluga.
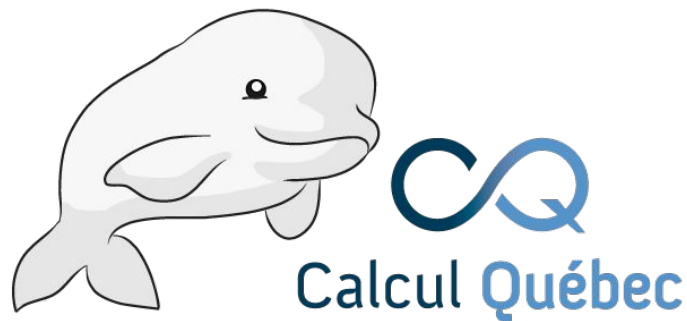
Workshop Outline

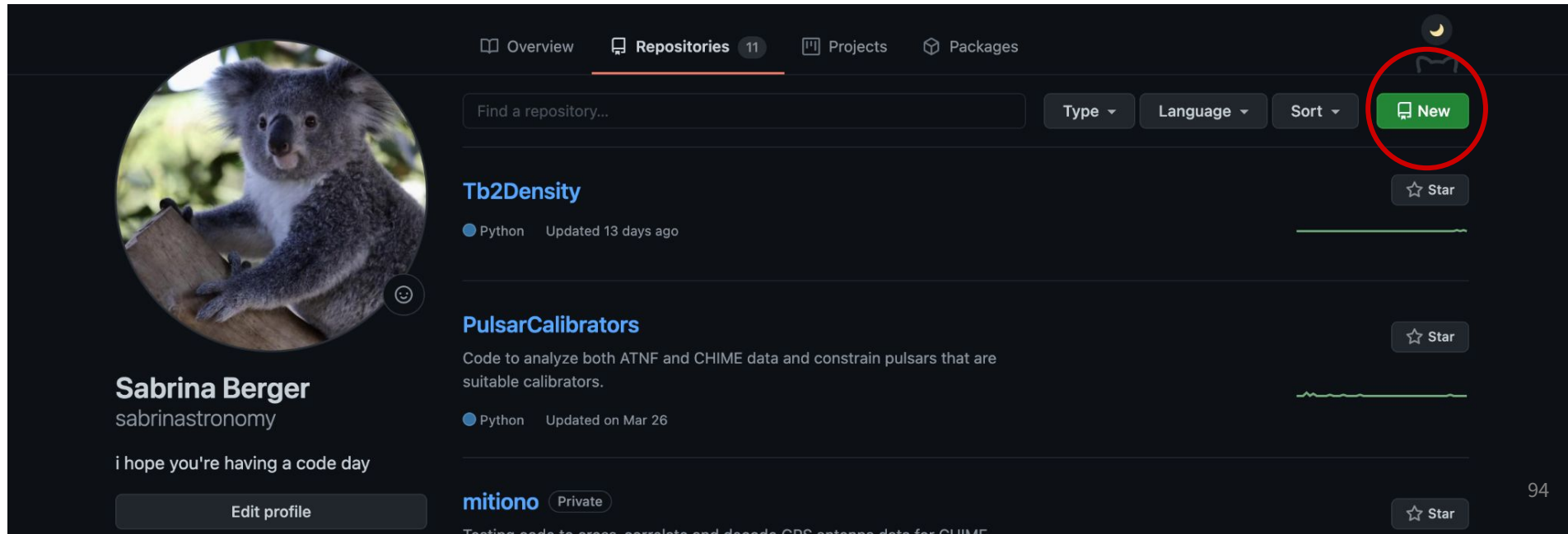Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- **What is git?**
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

What *is* Git?

- Git is what is known a Version Control System (VCS)

- By using git, you can keep track of **what** changed in a coding project, **when** it changed, **who** changed it, and **why** (if you're keeping good commit messages!)

- Particularly useful in collaborative projects, where multiple people are making changes at once. If anything in your changes conflicts (known as "**merge conflicts**"), changes can be made (sort of) gracefully

- In some ways, git is like a philosophy of how collaboration in code should be done

GitHub: working collaboratively online
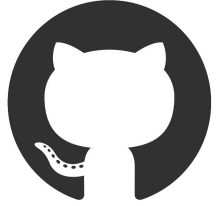
- GitHub is an online service for hosting projects managed with git online

- Expands on the branching of git with useful collaborative features

  - *Issues*

  - *Pull requests*

  - *Wikis, projects, …*

Pull requests (PRs)

| | | | Author ▾ | Label ▾ | Projects ▾ | Milestones ▾ | Reviews ▾ | Assignee ▾ | Sort ▾ |

⇅ 282 Open   ✓ 11,414 Closed

⇅ **step-between as drawstyle [Alternative approach to #15019]** ✕ `Needs rebase` `stale`   💬 136
#15065 opened on Aug 15, 2019 by andrzejnovak • Draft   ⊟ 6 of 6 ▭▭▭▭ ⬦v3.4.0

⇅ **wx backends: don't use ClientDC any more** ✕ `GUI/wx` `Needs rebase` `Needs revision`   💬 96
#11944 opened on Aug 26, 2018 by DietmarSchwertberger • Review required   ⊟ 0 of 6 ▭▭▭▭ ⬦v3.4.0

Note: *GitHub != git!*

- Very common misunderstanding
- *git* is the original VCS code, and doesn't have online hosting on its own
- *GitHub* is an online cloud hosting service with extra features for git projects

Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- **Tutorial 1: Set up your first git repository**
- Tutorial 2: Make a branch
- Tutorial 3: Check out an old version of your code

Tutorial 1: Making a repository

- Hopefully everyone's made a GitHub account to follow along with this tutorial -- if you haven't, you can quickly to follow along!

Follow the instructions to create a repository (don't worry about any of the extra features for now).

Now we want to *git clone* the repository we just made on to our local machines (wherever we'd like!)

Tutorial 1: Making a repository (Summary)

- We can **clone** (*git clone*) GitHub repos to our local machine, which will copy over all the git history from the online project (**./git** folder!)

  - **Note:** this makes a **remote branch** called **origin** which is attached to the cloud, this can be a confusing detail if you accidentally make a local branch and then try and merge it into origin

- We can also create a git repository locally (*git init*), which creates a **./git** folder locally. We can then push to GitHub later

  - IMO, this workflow is confusing, and I recommend starting *all* personal coding projects with repositories on GitHub

Important basic git commands

- git --help
  - Gives a helpful list of git commands! Can also do `git {command} --help` for specifics on commands
- git config --global user.name "{username}"
- git config --global user.email {email}
  - **Note:** these are *git* associated names/emails, so they don't have much to do with GitHub! Just useful for identifying yourself in local git projects
- git status
  - Shows you current changes
- git log
  - Shows you the commit history for your project
- git fetch
  - Updates information stored in the local **./git** folder, such as new remote branches

Making our first commit

- The most basic git workflow consists of three important steps:

  a. Use `git add {file}` to "add" a new file, or to "add" changes to an already existing file

     - You can use "wildcard" operators with this! E.g. `git add .*py` to add .py file changes

  b. Once you're happy with your additions, use `git commit -m "{useful message}"` to add a commit with a helpful commit message explaining your changes

  c. Finally, we need to push our changes to the cloud. We'll do this with the command `git push origin master`

     - Note that the *origin* here specifies the remote cloud, and *master* is the branch we're committing to. By default, GitHub repos start with only a master branch

Pulling from remote

- To pull in any changes from collaborators, just use `` `git pull` `` in the relevant directory

  - This won't do anything for us now, since we just made this repo for ourselves… But it's VERY important to pull the most recent version of the repository before you start making changes!

  - Ideally if everyone was working on their own branches and being responsible about workflow, this wouldn't be an issue… But nobody's perfect :)

Summary

- Step 0: PULL changes that might have been made by collaborators

- Step 1: ADD our changes

- Step 2: COMMIT changes with a message (-m)

- Step 3: PUSH changes to the cloud

- **NOTE:** all the while we can check the STATUS of our additions!

Exercise 1:

1. Create a GitHub repository on your account
2. Move the files from *EIEIOO_Scripts* to the cloned repository and make your first commit.
3. Then change test2.py to print "One day Github will save me from the coding monsters".
4. Push your changes to GitHub

Workshop Outline
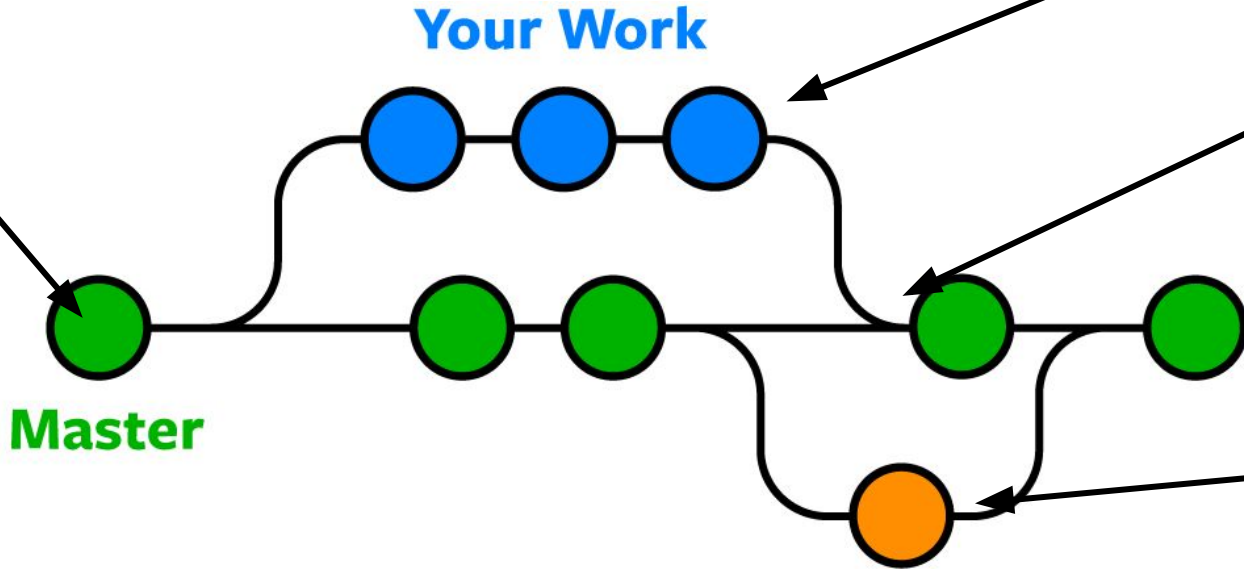
Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- **Tutorial 2: Make a branch**
- Tutorial 3: Check out an old version of your code

Tutorial 2: Make a Git branch

You want to code a new feature in the code without breaking the stable code base, so you make your own branch!

Your Work

Start with some original version of the code base

Merge your changes into the master branch once they're stable!

Master

Others can write features in tandem!

Someone Else's Work

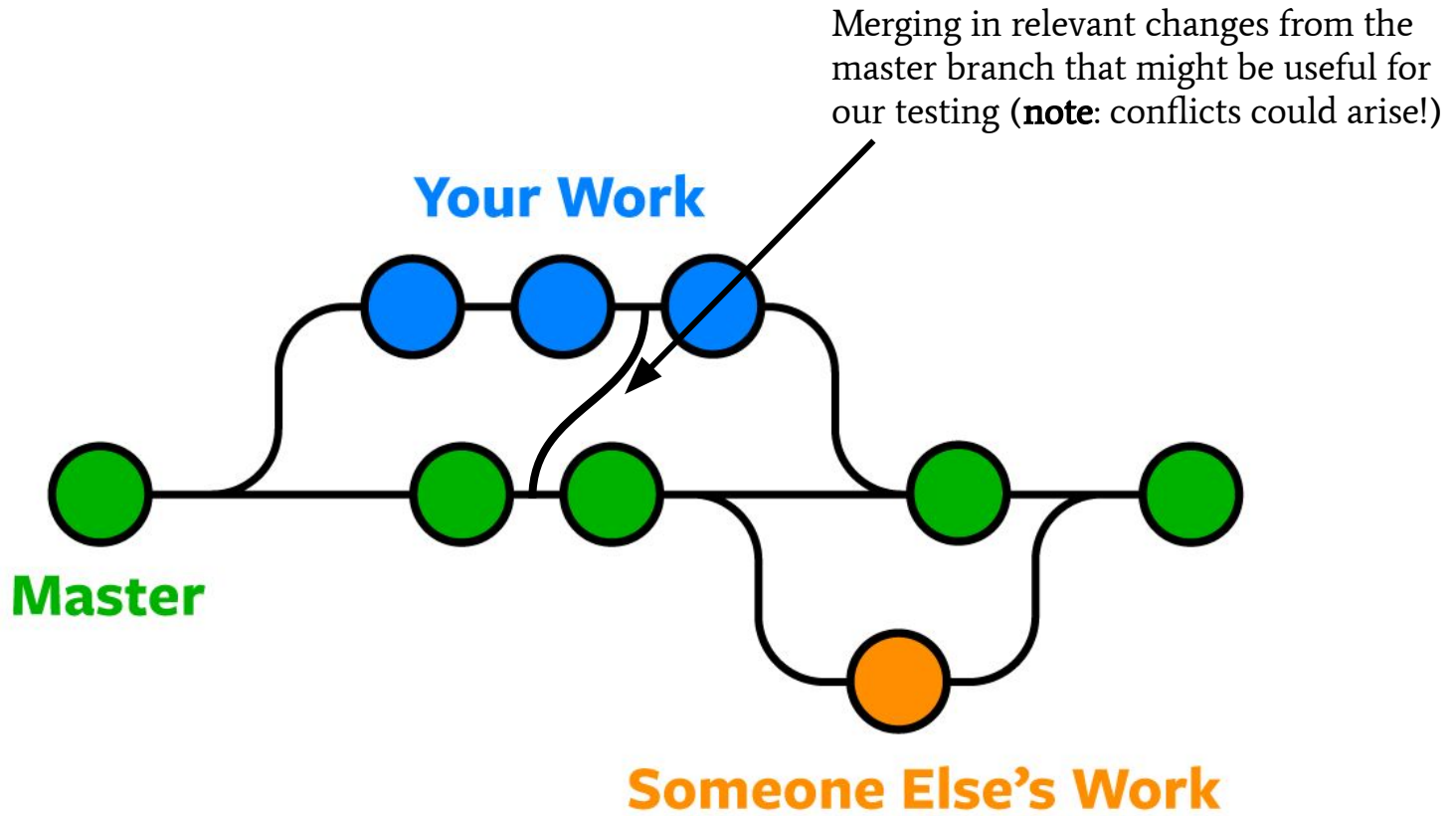# Making our own branch

- We can take a look at the available branches with `git branch`

  - Can do `git branch -r` for remote branches, `git branch -a` for all branches

- Make our own branch with `git branch {branch name}`

- NOTE: `**git branch {branch name}**` only makes a branch *locally!* Later, we'll see how to get this branch on GitHub

- The branch command only *made* the branch. Now if we want to ***checkout*** to our new branch -- i.e. move to the space where we'll make our changes
  - `**git checkout {branch name}**`
  - **NOTE:** we can checkout to a new branch in one command! `**git checkout -b {new branch}**`

- Now we can safely make our changes without interfering in the stable master branch!

- Once we've ADDED and COMMITTED our changes, we can PUSH!
  - **NOTE:** we have to do `**git push --set-upstream origin {our branch}**` -- this is because our branch has only been local, until now: we're making our branch sync with the cloud with **--set-upstream**

Merging in changes from other branches

- Usually a good practice to compare differences between branches first

    - `git diff {one-branch} {other-branch}` to compare

- Now, say we're working on our own feature branch, and there are some useful changes on another part of the code base in **_origin/master_** we want on our branch

    - First: `git pull` to update your local branches with changes from the cloud (**origin**)

    - Next: `git merge {other-branch}` to put those changes in your current branch!

Merging in relevant changes from the master branch that might be useful for our testing (**note**: conflicts could arise!)

Your Work

Master

Someone Else's Work

Exercise 2: Make your own branch

Create and push a new branch to your online GitHub repository for EIEIOO_Scripts. Call the branch "trying_new_things".

Making a pull request

- When you're working in a collaboration, and you're ready to incorporate your changes into the master branch, you can make a pull request!

- Pull requests are a super useful way of keeping major code changes organized

- Rule of thumb: **master** branch should ALWAYS be deployable

  - This is why pull requests exist: typically if you're a part of a collaboration, there will be other people working on the code base with you. Usually there'll be one/a few people who manage most of a given repository, and making a pull request allows you to give them a chance to view your code, review it, suggest changes, and then finally accept the merge into master once it's deemed ready

- Workflow goes something like

  - Propose a new feature

  - Checkout a new branch to start working on your feature (make sure nobody interferes with your work so you don't get merge conflicts)

  - Keep pushing changes to your branch until things are stable/finished, then make a pull request

Dealing with merge conflicts

- Despite best efforts to keep organized, issues *will* arise!

- Merge conflicts are the part of git that will, at some point in your coding life, make you scream at your computer

- We can fix these problems pretty easily, in fact! Try not to resort to saving your changes locally, and re-downloading the whole repo
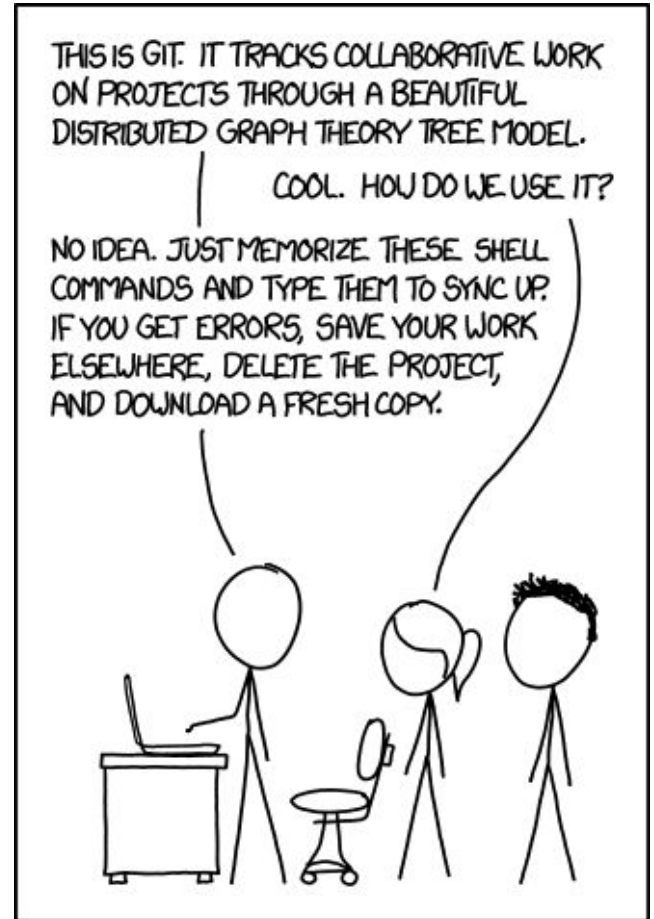
Workshop Outline

Part 1: Learn the basics of the command line (CLI)

- What is Linux?
- Tutorial 1: Navigating the Filesystem
- Tutorial 2: File Permissions
- Tutorial 3: The Builtin Software Library

Part 2: Understand how to use *git*

- What is git?
- Tutorial 1: Set up your first git repository
- Tutorial 2: Make a branch
- **Tutorial 3 (Final): Checkout an old version of your code**

Going back to an old commit when your new code breaks!

- You can get a log of all previous commits with
  - `git log`

- This should return your previous commits along with their corresponding hash, e.g.,

```
[(base) sabrinaberger@sabrinastronomy summerschool % git log
commit 5d29d753054e787005f0202364e286d1211032db (HEAD)
```

- You can revert to a previous commit with
  - `git checkout <commit hash> . `

```
$ cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

The important parts of a merge conflict will show up in our conflicted files:

- <<<<<<< HEAD

- =======

- >>>>>>> new_branch_to_merge_later

Think of the "=======" as the conflict divider. The content between HEAD and the divider is our content, and the content between the divider and the new_branch_to_merge_later is the content we tried to merge in. By reconciling the differences on these lines of code in a text editor, once you're happy with the outcome, you can add/commit/push as usual!

Final Exercise

Restore your local *EIEIOO_Scripts* to its original commit before you modified the file.

Great job on all the tutorials today!