PLEASE FILL IN THIS → side of the classroom first!

If you are using one of the Gilman desktops…

- Sign in on a desktop

- Double click the desktop folder "SSH & Secure File Transfer"

- Double click on "putty.exe"

- Sign into the server

- Chill

If you need help, raise your hand or put an ORANGE sticky note on your monitor

Advanced UNIX

BCBGSO Workshop

March 3rd, 2018

Presenter: Carla Mann

Thanks!

- BIG thanks to Jennifer Chang for inspiration for slides and materials
- Organizers: Urminder Singh and Paul Villanueva
- Funding/Support/Volunteers: BCBGSO
- Tech support: Biology IT, especially Levi Baber

Our many, MANY volunteers:

Ashish Jain	Bekah Starks	Sagnik Banerjee
Sharmistha Chakrabortty	Yulu Chen	Basil Khuder
Avani Khadilkar	Weija Su	Hylia Gao
Alvin Chon		

Materials

- All exercise activities from this workshop are available at:

https://github.com/cmmann/20180302-unix-adv

- Supporting materials are available at:

https://github.com/cmmann/20180302-UNIX-ADVANCED-MATERIALS/

You can download this PowerPoint and follow along on your computer.

- You will probably benefit quite a bit from downloading (and using) the cheat sheet!

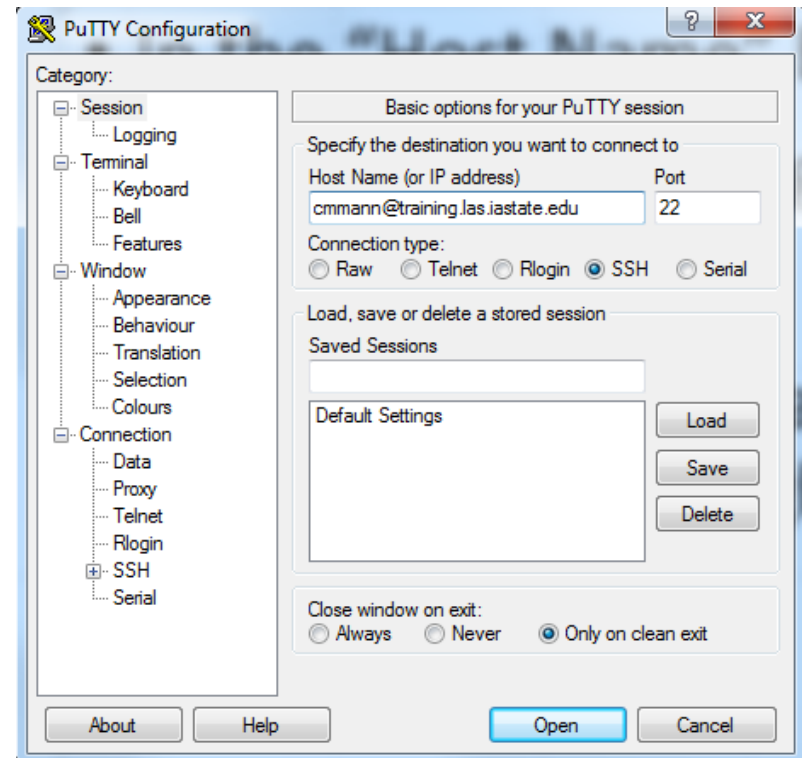# Set-Up

Mac/Linux:

Open terminal

```
ssh <your-
netid>@training.las.iastate
.edu
```

Windows:

Open Putty.exe

Enter

```
<your-
netid>@training.las.iastate
.edu
```
into the Host Name box

Overview

Lesson 0: Quick Review of Basic UNIX

Lesson 0.5: Setup

Lesson 1: Text Editing with `nano`

Lesson 2: Shell Scripting

Lesson 3: Condo and Slurm

Lesson 4: Data Exploration with `grep`

Lesson 5: Regular Expressions (if we get to it)

Lesson 0: Quick Review

- When describing a path to an/application:
  `this/is/path/to/the/file.txt`
- For our purposes:
  - "folder" and "directory" refer to the same thing
  - "terminal", "console", and "console window" all refer to the place you will type commands
- In PowerPoint, commands you will type in the terminal will look
  `like this`
- Keys you press will look like this: **[Ctrl]** or **[command]**
- If you should press keys at the same time: **[Ctrl]** + **[C]**
- A name or value that is user-dependent or variable will look `<like this>`
- Don't use spaces in names; use dashes **[-]** instead

`cd`: change directory

`ls`: list directory contents

`man <command>`: show manual page for command

Lesson 0.5: Setup

Once logged in, use this command to clone the materials into your workspace:

```
git clone
https://github.com/cmmann/20180303-unix-
adv.git
```

If you are prompted for a username/password, check to make sure you typed the URL correctly!

Lesson 1: Text Editing

Overview:

Lesson 1.1: Text Editors in UNIX

Lesson 1.2: `nano`

Lesson 1.1: Text Editors in UNIX

Multiple ways of editing text in files in UNIX

`Vim` is a VERY powerful text editor, but has a steep learning curve
- Very worthwhile to learn, but we could spend an entire workshop on it, so we're not going to mess with it today

"Friendliest" Unix text editor is `nano`

*If you already know how to use Vim, feel free to use it!
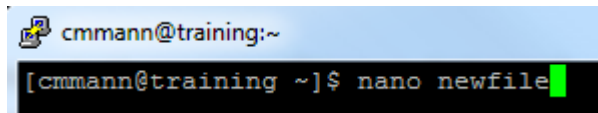
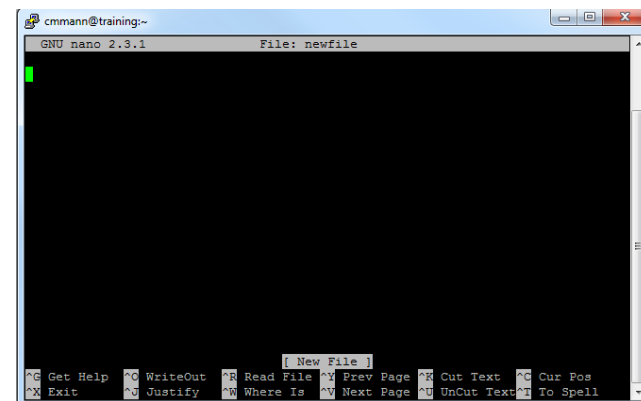# Lesson 1.2: Text Editing with nano

Command:

`nano <filename>`

What it does:

If `<filename>` exists, `nano` will open the file and you can read and manipulate it

If `<filename>` does not exist, `nano` will create a new file called `<filename>` and open it for you
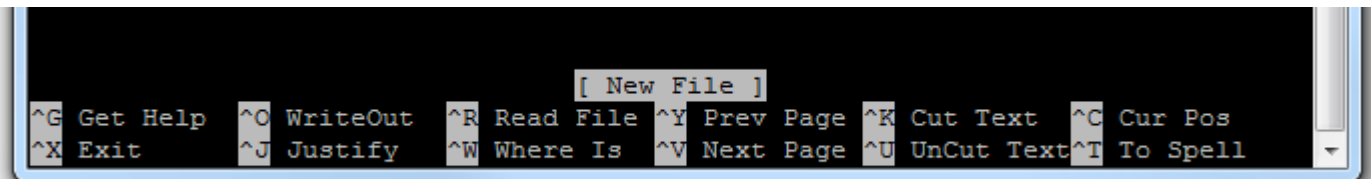
Lesson 1.2:
Text Editing with `nano`

`nano` is kind enough to give you a list of controls at the bottom of the editing window

You can easily type in the `nano` window just as you would in Notepad, TextEdit, or any other text editor

To exit out of a `nano` window, type **[Ctrl]+[x]**

`nano` will ask if you want to save changes; type **[y]** or **[n]**

```
                          [ New File ]
^G Get Help   ^O WriteOut   ^R Read File ^Y Prev Page ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Exercise 1:

Goals:

1. **Navigate** to `adv-unix/exercise1/`
2. Use **nano** to open "`exercise1.txt`"
3. **Edit** the text of "3. What is the capital of Assyria?" to read "3. What is your favorite color?"
4. Answer the questions in the file
   (the answers don't really matter, just your ability to edit the file)
5. **Exit** (save when prompted)

Exercise 1:

Goals:

1. Navigate to `adv-unix/exercise1/`

2. Use nano to open "`exercise1.txt`"

3. Edit the text of "3. What is the capital of Assyria?" to read "3. What is your favorite color?"

4. Answer the questions in the file
   (the answers don't really matter, just your ability to edit the file)

5. Exit (save when prompted)

Exercise 1:

Goals:

1. Navigate to `adv-unix/exercise1/`

2. Use nano to open "`exercise1.txt`"

3. Edit the text of "3. What is the capital of Assyria?" to read "3. What is your favorite color?"

4. Answer the questions in the file
   (the answers don't really matter, just your ability to edit the file)

5. Exit (save when prompted)

Exercise 1:

Goals:

1. Navigate to `adv-unix/exercise1/`

2. Use nano to open "`exercise1.txt`"

3. Edit the text of "3. What is the capital of Assyria?" to read "3. What is your favorite color?"

4. Answer the questions in the file
(the answers don't really matter, just your ability to edit the file)

5. Exit (save when prompted)

Exercise 1:

Goals:

1. Navigate to `adv-unix/exercise1/`

2. Use nano to open "`exercise1.txt`"

3. Edit the text of "3. What is the capital of Assyria?" to read "3. What is your favorite color?"

4. Answer the questions in the file
(the answers don't really matter, just your ability to edit the file)

5. Exit (save when prompted)

Lesson 2: Shell Scripting

Overview:

Lesson 2.0: What is a shell?

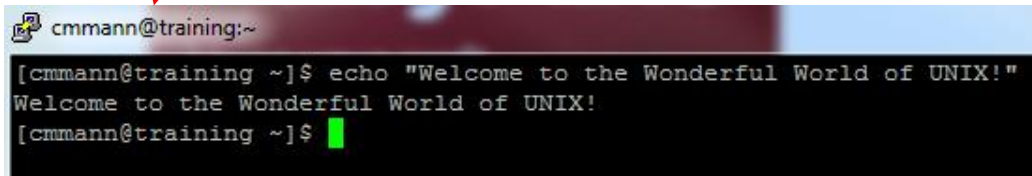Lesson 2.1: Creating a shell script

Lesson 2.3: Executing a shell script
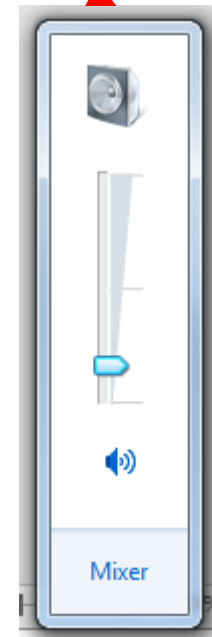
Exercise 2: Completed throughout the lesson

Lesson 2.0: What is a shell?

A shell is an interface for accessing an operating system's services

Shells can be GUIs (graphical user interface)

or CLIs (command-line interface)

```
cmmann@training:~
[cmmann@training ~]$ echo "Welcome to the Wonderful World of UNIX!"
Welcome to the Wonderful World of UNIX!
[cmmann@training ~]$
```

Mixer

Lesson 2.0: What is a shell?

There are multiple 'flavors' of command-line interfaces:

       DOS, POSIX, CMD.EXE, many others

We are going to use a command-line shell
called Bash:

To enable Bash scripting on your terminal, enter:

`bash`

And as simply as that, the server now knows what to use to
interpret commands

Lesson 2.0: Shell Scripting

Every command you type into your terminal can be saved into a file

This file is called a *shell script*

That file can then be *executed*, or run, from the terminal

The commands in the file will be read line-by-line and executed, as if you had typed them in the terminal

Lesson 2.1: Creating a Shell Script

When creating a shell script, we first need to create a file

This file should end in ".sh", which signifies that it is a shell script

Note that the computer doesn't require the ".sh" extension to recognize this (it uses something different) – this is a human convention so you know the file contains a shell script

Exercise 2:

Create a file, using `nano`, called "`hello.sh`"

Lesson 2.1: Creating a Shell Script

WE know the script should be executed with `bash` – Unix doesn't.

How do we tell UNIX what tools to use?

By starting off the file with a hashbang and a file path!
This tells UNIX to use certain a certain shell to run the script

```
#!<path/to/program>
```

In our case, we're using bash. So the first line of `hello.sh`, and EVERY SHELL SCRIPT YOU WRITE (that will be interpreted with bash), will be:

`#!/bin/bash`

Exercise 2: Go ahead and add this header to `hello.sh`

Lesson 2.1: Comments

Scripts can be complicated

Keep track of what scripts are doing with *comments*

In .sh files, any text following '#' is ignored by bash

#So this is ignored
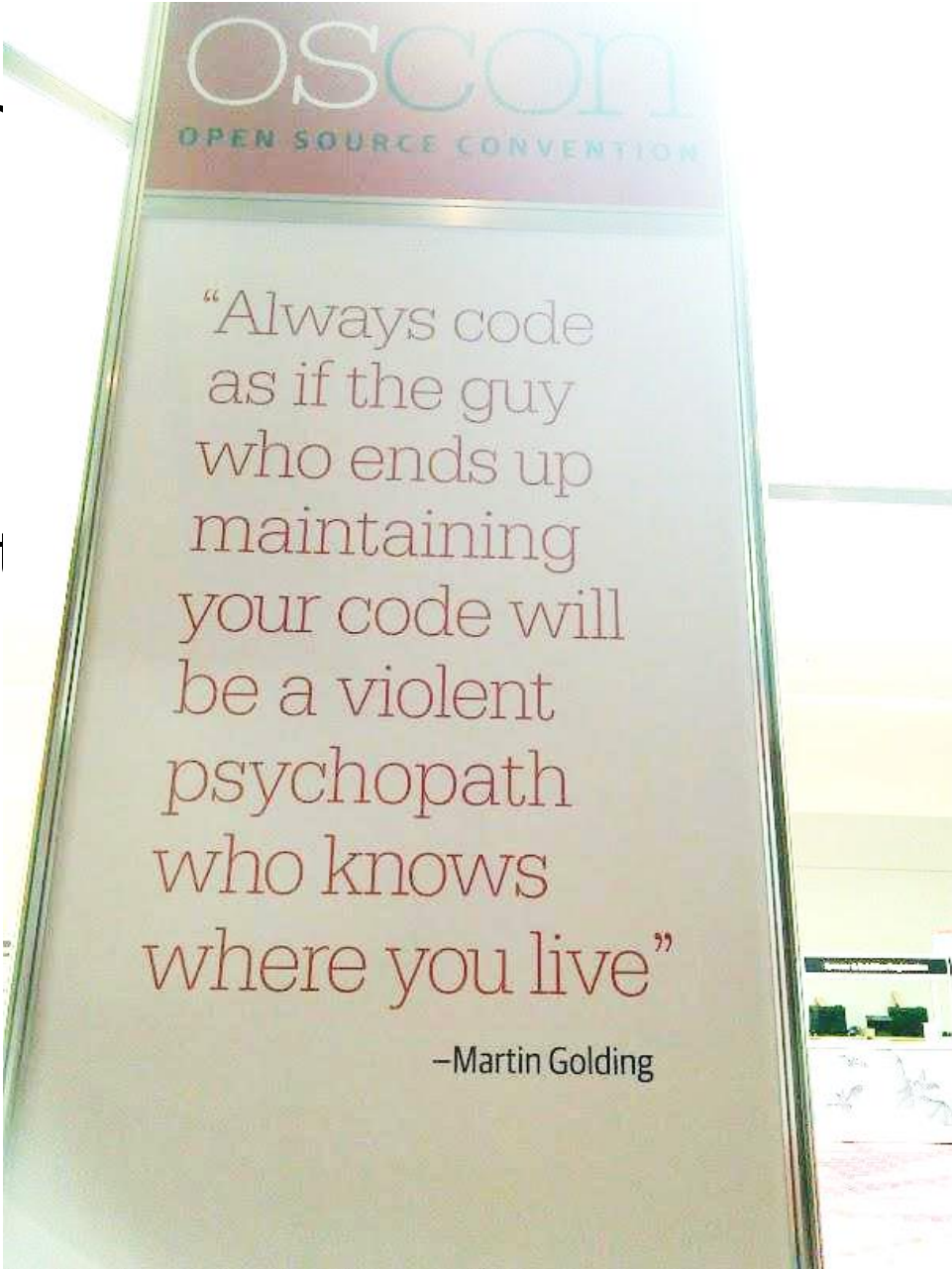But only #this last bit is ignored

Exercise 2:

#Using a comment, add your name and the date to your `hello.sh` script

Lesson

It is good   s well

Will you st   g 6 months
from now

OSCON
OPEN SOURCE CONVENTION

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live"

–Martin Golding

Lesson 2.1: Creating a Shell Script

We are going to create a simple script within `hello.sh` that prints "hello" to the console upon execution.

You can print to the console using the command "echo":

```
echo "what you want to say"
```

Lesson 2.1: Creating a Shell Script

You can do the same thing, within the script

Exercise 2:

In your hello.sh file, type:

```
echo "hello world!"
```
or
```
echo "Hello, world!"
```
 (if you want to be grammatically correct)

Then exit and save the file.

Lesson 2.2:
Executing a Shell Script

You can execute scripts you've written (that are in your present working directory) by typing:

`./scriptname.sh`

This tells the server the path to the command it's executing

But we execute other commands by typing just `ls` or `cd` or `echo`

Why can't we just execute the file by typing it's name?

Lesson 2.2:
Executing a Shell Script

Security.

What happens if somebody comes into your directory and creates an executable file called `ls` that contains:

```
#!/bin/bash
echo "sucks to be you"
rm -rf /
```

(Don't create this file or run `rm -rf /`)

Lesson 2.2: Executing a Shell Script

Security.

What happens if somebody comes into your directory and creates an executable file called `ls` that contains:

```
#! /bin/bash
echo "sucks to be you"
rm -rf /
```

This way, you can be sure that you're using the genuine `ls` command.

Lesson 2.2:
Executing a Shell Script

So how do we execute `hello.sh`?

Lesson 2.2:
Executing a Shell Script

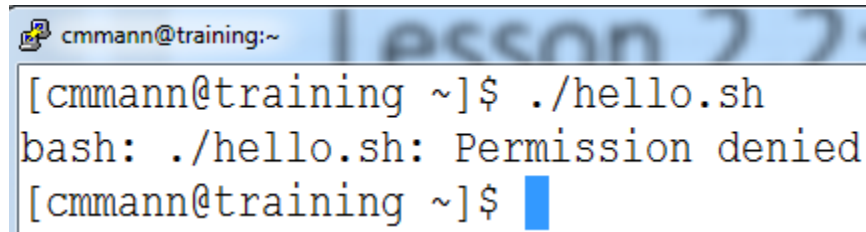So how do we execute `hello.sh`?

What happens when you run the script?

Lesson 2.2: Executing a Shell Script

So how do we execute `hello.sh`?

What happens when you run the script?

Lesson 2.2:
Executing a Shell Script

See what happens with:

```
bash hello.sh
```

Lesson 2.2:
Executing a Shell Script


```
cmmann@training:~
[cmmann@training ~]$ ./hello.sh
bash: ./hello.sh: Permission denied
[cmmann@training ~]$ ls -l hello.sh
-rw-r--r--. 1 cmmann domain users 40 Mar  1 22:44 hello.sh
[cmmann@training ~]$ bash hello.sh
hello, world!
[cmmann@training ~]$
```

Why is the script executing, even though we don't have permission!?!?

Lesson 2.3:
Executing a Shell Script

Fun fact: The execute permission is not a *security* feature – instead, it's a flag to the system that a script is executable, and the system now knows to look for a `#!` header line to know what program to use to *interpret* the instructions in your script.

So when we run a script with `./hello.sh`, we are executing *hello.sh*, which the system does not recognize as executable

When we run the script with `bash hello.sh`, we are executing *bash*, which is reading `hello.sh` and then executing the commands it has read
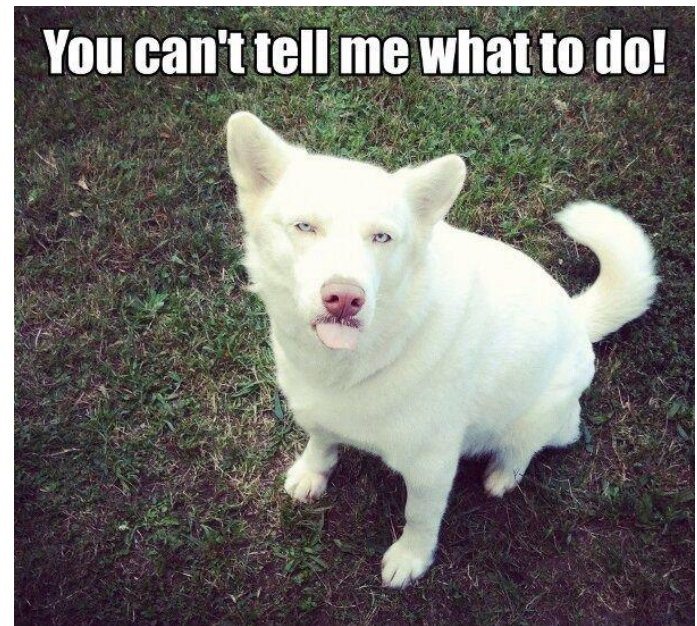
Lesson 2.3: Executing a Shell Script

The difference between the two:

In scenario 1, `hello.sh` is telling the system what to do.

In scenario 2, `bash` is reading `hello.sh`, **and then** `bash` is telling the system what to do.

`bash` **has executable permissions,** so it can 'boss' the system around, **but** `hello.sh` **currently doesn't,** so it can't.

You can't tell me what to do!

Lesson 2.3:
Executing a Shell Script

Try changing the permissions on `hello.sh` to make it executable for you, the owner

How would you change the Execute permission for hello.sh?

`chmod ??? hello.sh`

Remember: read = 4, write = 2, execute = 1

Lesson 2.3:
Executing a Shell Script

Exercise 2:

Try changing the permissions on hello.sh to make it executable for you, the owner

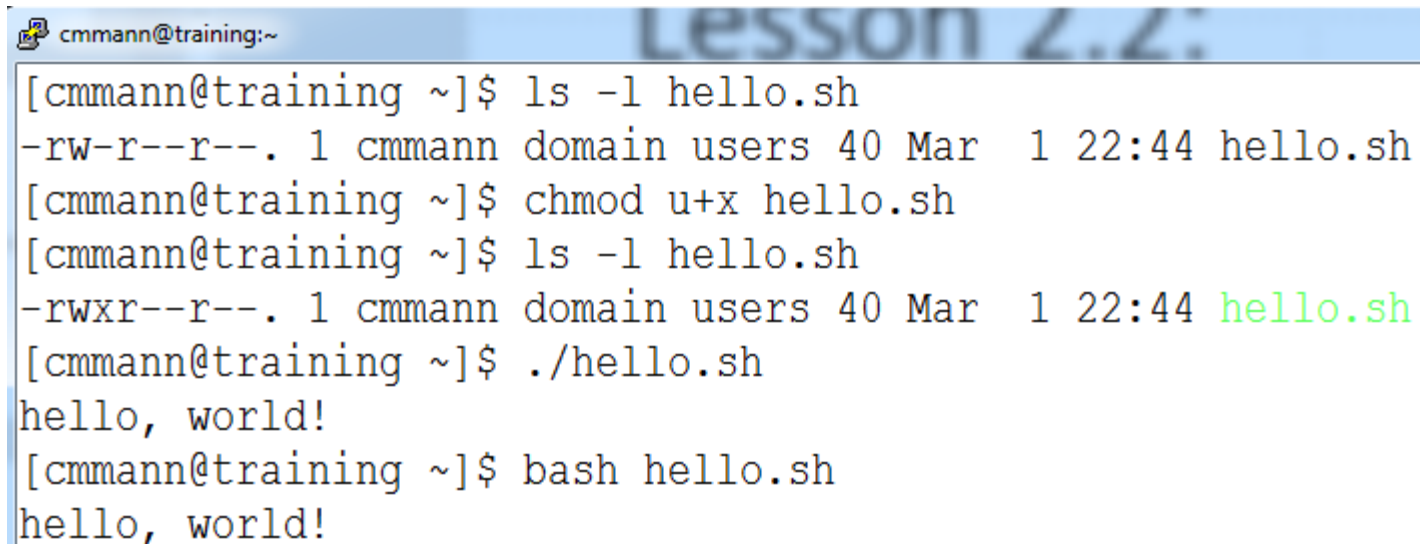How would you change the execute permission for hello.sh?

```
chmod ??? hello.sh
```

Remember: read = 4, write = 2, execute = 1

Lesson 2.3:
Executing a Shell Script

Try executing the script now:

`./hello.sh`


```
cmmann@training:~
[cmmann@training ~]$ ls -l hello.sh
-rw-r--r--. 1 cmmann domain users 40 Mar  1 22:44 hello.sh
[cmmann@training ~]$ chmod u+x hello.sh
[cmmann@training ~]$ ls -l hello.sh
-rwxr--r--. 1 cmmann domain users 40 Mar  1 22:44 hello.sh
[cmmann@training ~]$ ./hello.sh
hello, world!
[cmmann@training ~]$ bash hello.sh
hello, world!
```

Lesson 2.3:
Executing a Shell Script

If Execute permission isn't providing security, then what is?

Lesson 2.3:
Executing a Shell Script

If Execute permission isn't providing security, then what is?

Read and Write permissions!

Try changing permission of `hello.sh` so that you have Write and Execute, but not Read:

Lesson 2.3:
Executing a Shell Script

If Execute permission isn't providing security, then what is?

Read and Write permissions!

Try changing permission of `hello.sh` so that you have Write and Execute, but not Read:

Lesson 2.3:
Executing a Shell Script


```
cmmann@training:~
[cmmann@training ~]$ ls -l hello.sh
-rwxr--r--. 1 cmmann domain users 40 Mar  1 22:44 hello.sh
[cmmann@training ~]$ chmod 344 hello.sh
[cmmann@training ~]$ ls -l hello.sh
--wxr--r--. 1 cmmann domain users 40 Mar  1 22:44 hello.sh
[cmmann@training ~]$ ./hello.sh
/bin/bash: ./hello.sh: Permission denied
[cmmann@training ~]$ bash hello.sh
bash: hello.sh: Permission denied
```

Lesson 2.3:
Executing a Shell Script

Why don't they work?

Lesson 2.3:
Executing a Shell Script

Why don't they work?

Because without Read permission, the system can't read the commands in the file, regardless of how it's called!

So Read (and to a lesser extent, Write) permissions are the true 'security' features of permissions

Exercise 2:

Goal:

1. Execute `hello.sh` by calling

`./hello.sh`

Lesson 3: Condo and Slurm

Overview:

Lesson 3.1:
High Performance Computing

- Also known simply as "HPC"

> " High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.
>
> https://insidehpc.com/hpc-basic-training/what-is-hpc/

Typically, an HPC resource consists of a "cluster" of computational nodes, so sometimes you will hear HPC referred to as a "cluster"

http://www.ticra.com/products/software/grasp/mpi-hpc-linux-cluster

Lesson 3.1: High Performance Computing

- Many people can simultaneously use an HPC cluster

- An HPC resources can perform several computationally intensive jobs at once by distributing those jobs across several computational nodes
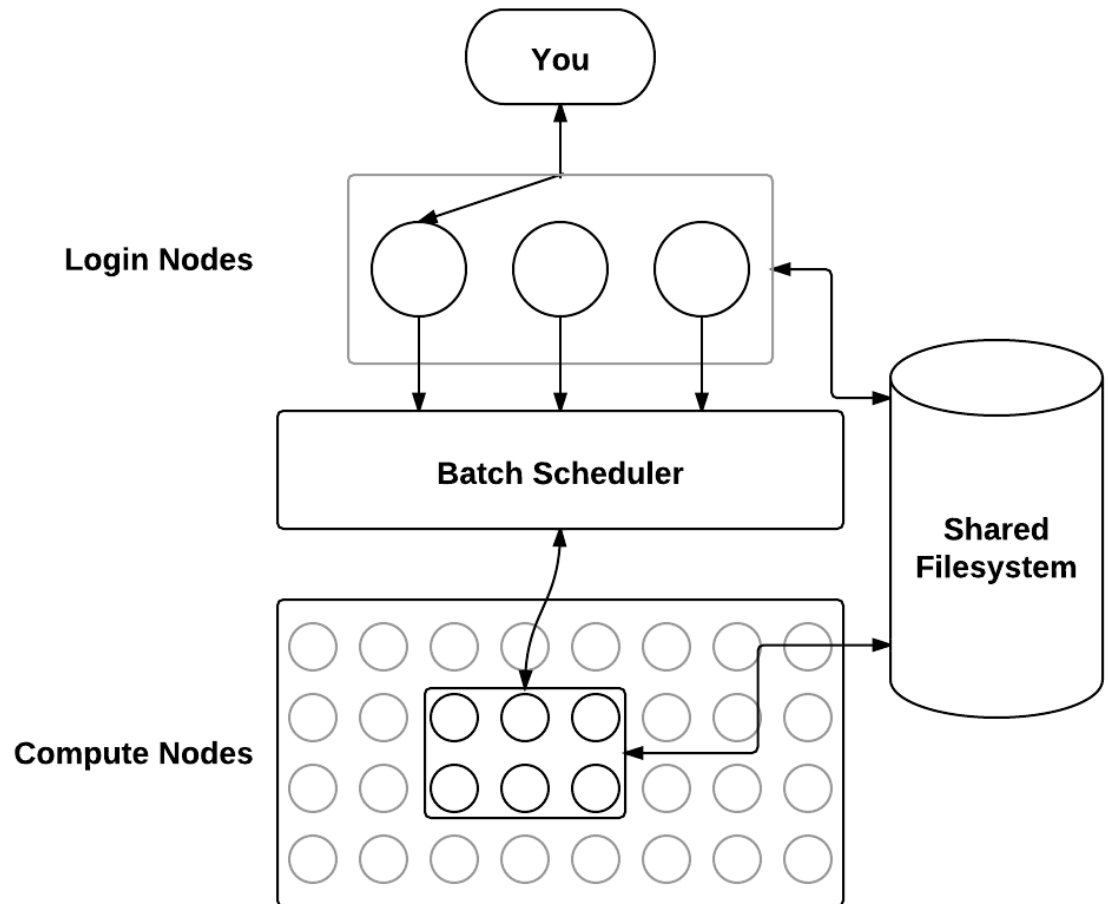
Lesson 3.1:
High Performance Computing

- Users communicate with the server through SSH, and connect to a "head" or "login" node
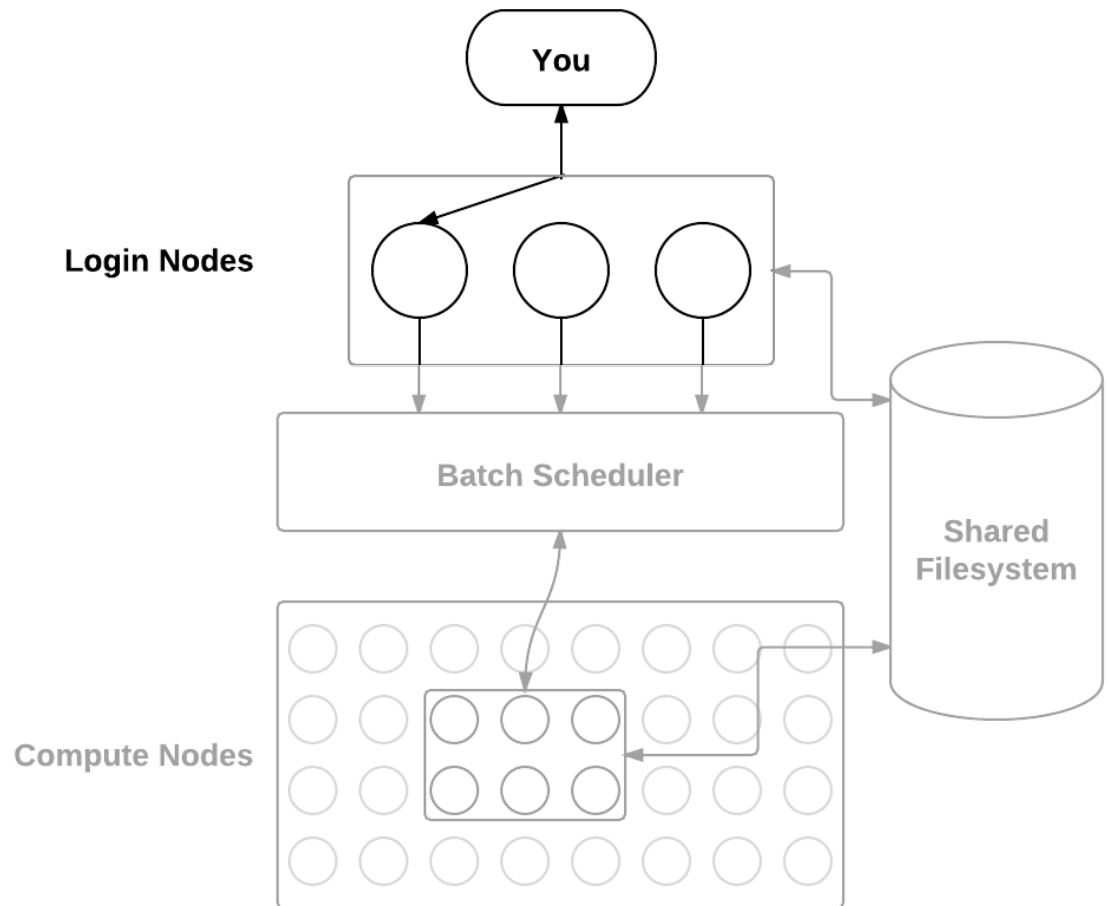
Lesson 3.1:
High Performance Computing

- The user submits a job through the head node to a "scheduler"

You

Login Nodes

Batch Scheduler

Shared Filesystem
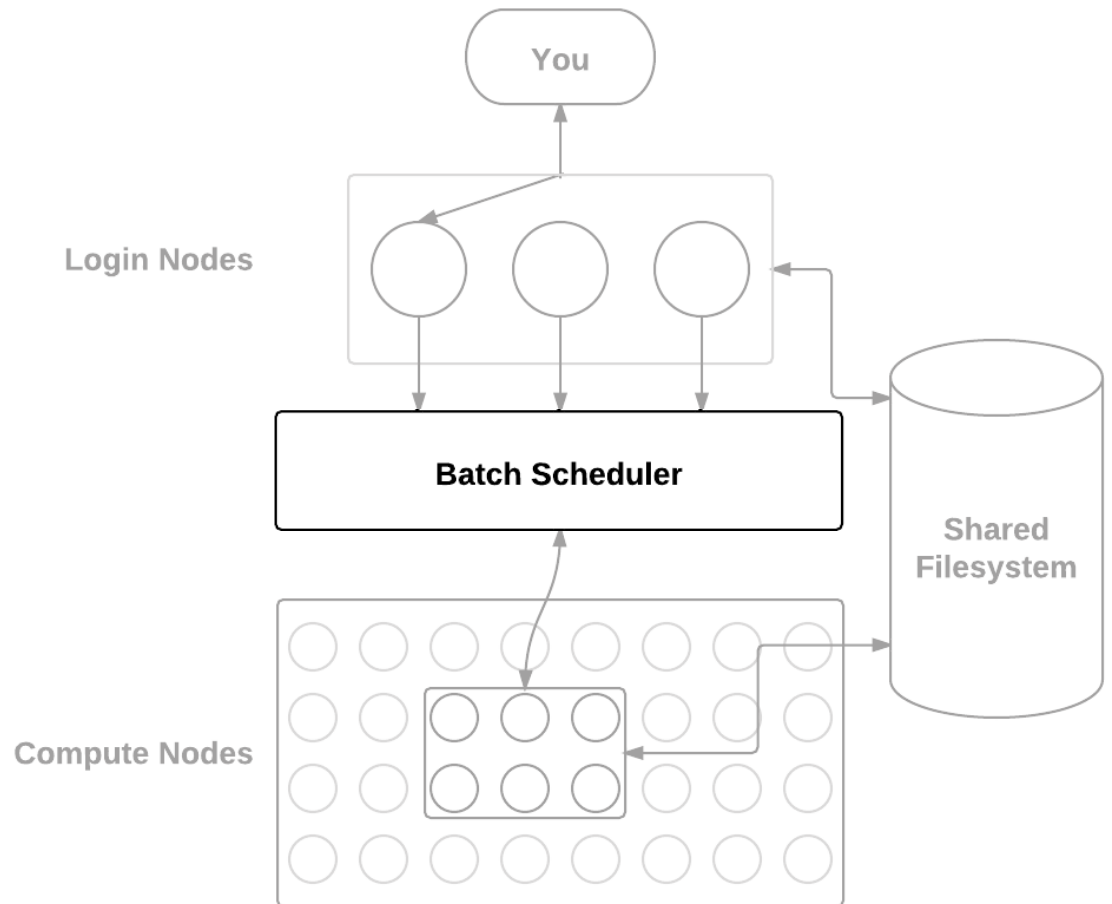
Compute Nodes

Lesson 3.1:
High Performance Computing

- The scheduler distributes jobs across the "compute" nodes in the cluster, so that jobs can be performed as efficiently as possible

You

Login Nodes

Batch Scheduler

Compute Nodes

Shared Filesystem
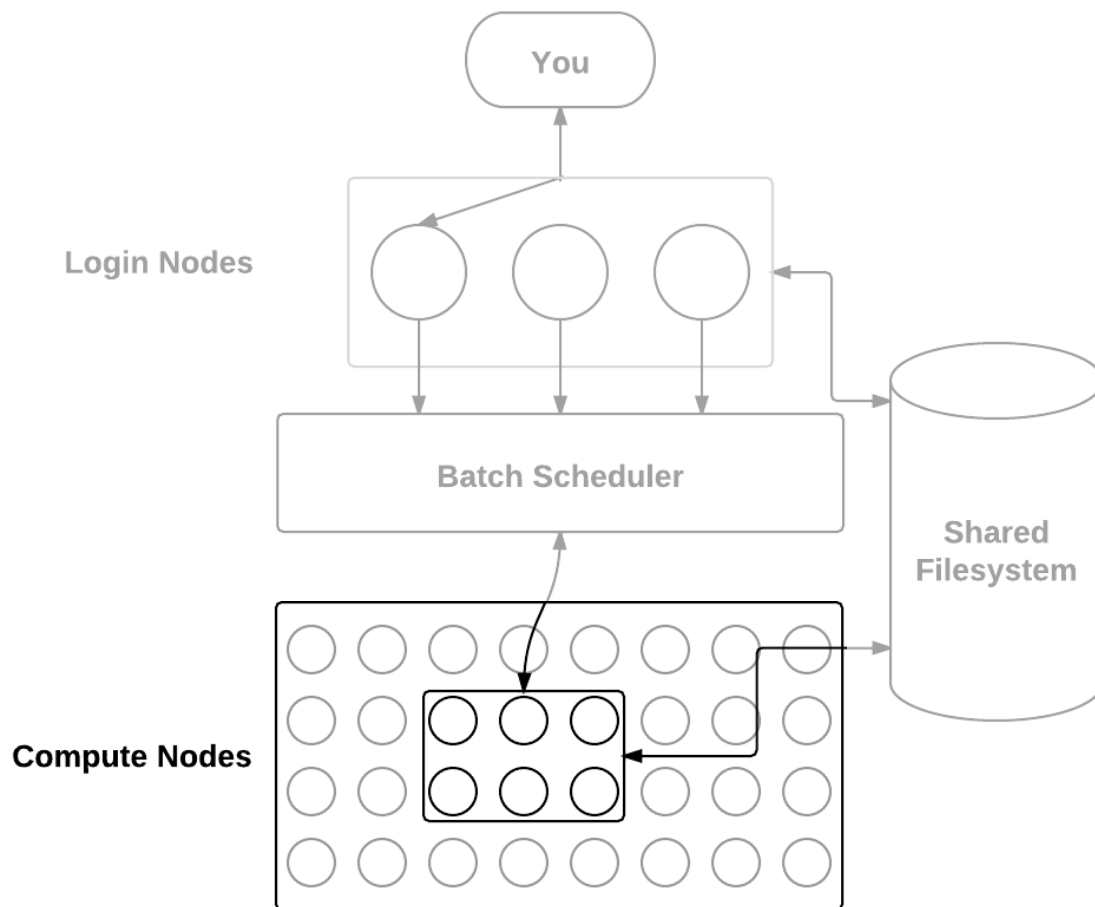
Lesson 3.1:
High Performance Computing

- Once the compute nodes have finished performing a job, the results are stored where the user can access them
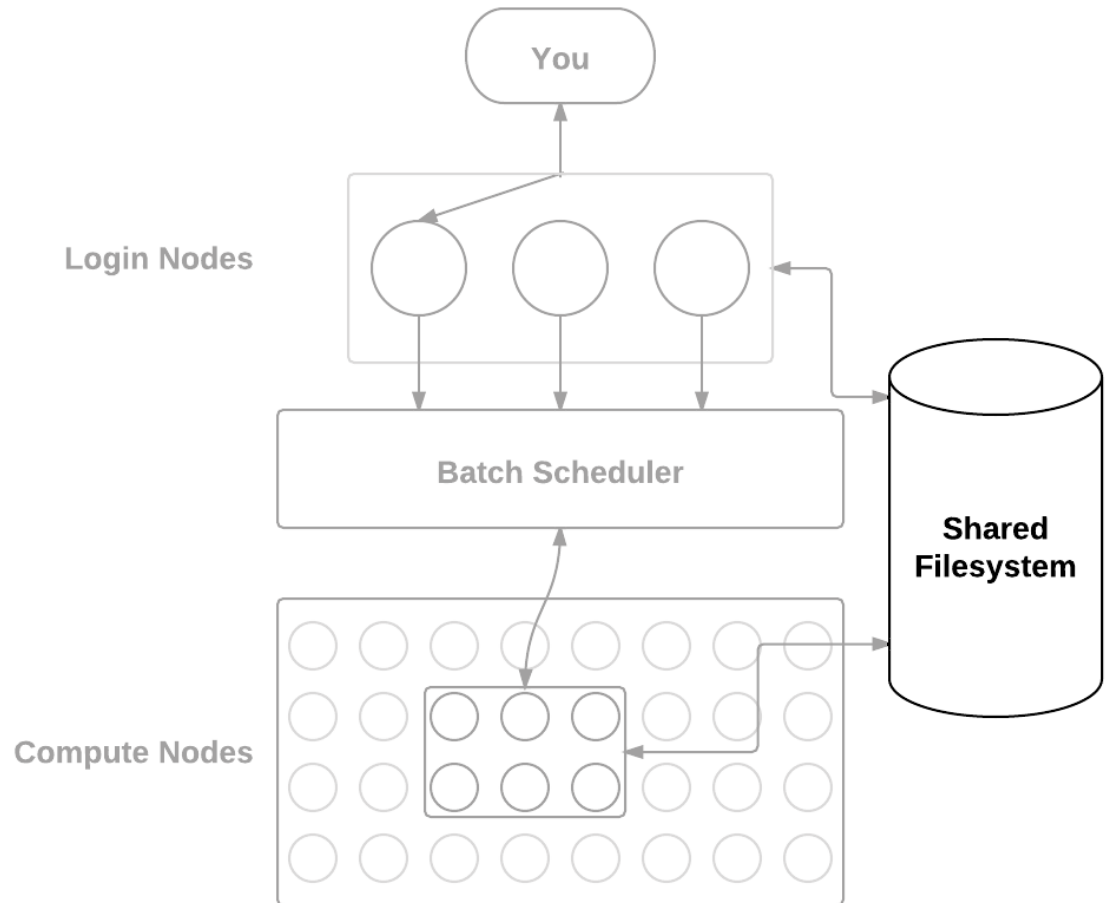
Lesson 3.2: Condo

- Condo is one of ISU's frequently utilized HPC resources

- Further information: https://www.hpc.iastate.edu/guides/condo-2017

- ITS has very good guides on IT resources

Lesson 3.2: Condo

- NEVER DO LARGE OPERATIONS DIRECTLY IN YOUR CONSOLE/ON THE HEAD NODE

- This means you shouldn't run large jobs directly in your terminal when connected to an HPC cluster

- When SSH'd into the HPC cluster, **the environment you're in is only meant to handle task scheduling**

- You submit jobs to the compute nodes through Slurm scripts

Lesson 3.3: Slurm

- Simple Linux Utility for Resource Management

- The name is indeed referencing Slurm "soda" from the TV show *Futurama*

Lesson 3.3: Slurm

- Slurm is the batch scheduler that ISU uses for distributing jobs on our HPC resources
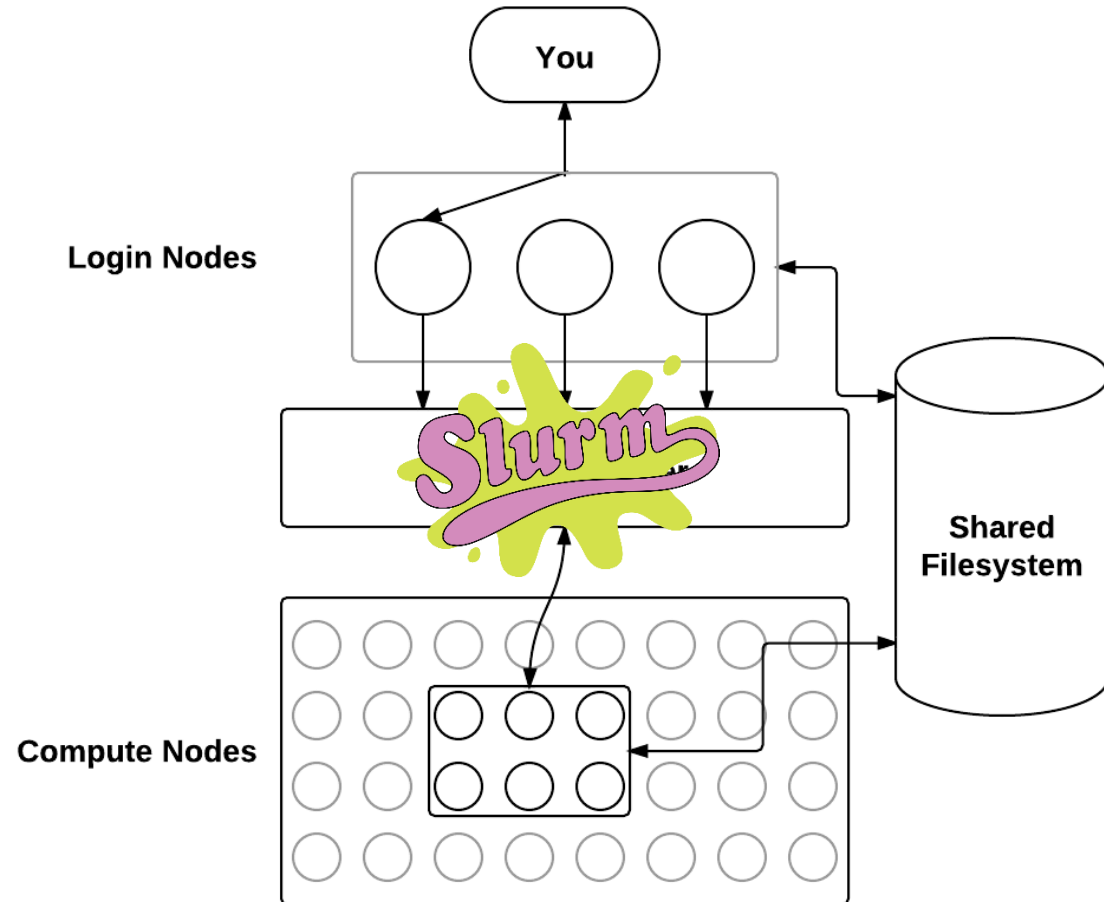
Lesson 3.3: Slurm

- HPC jobs are submitted through the console, in the form of scripts

- Each of these scripts will have special instructions for Slurm, and Slurm will handle them appropriately

```
#!/bin/bash

# Copy/paste this job script into a text file and submit with the command:
#    sbatch thefilename

#SBATCH --time=1:00:00   # walltime limit (HH:MM:SS)
#SBATCH --nodes=1   # number of nodes
#SBATCH --ntasks-per-node=16   # 16 processor core(s) per node
#SBATCH --job-name="Sample Job"
#SBATCH --mail-user=cmmann@iastate.edu   # email address
#SBATCH --mail-type=BEGIN
#SBATCH --mail-type=END
#SBATCH --mail-type=FAIL
#SBATCH --output="2018samplejob.txt" # job standard output file (%j replaced by job id)
#SBATCH --error="2018sampleerror.txt" # job standard error file (%j replaced by job id)

# LOAD MODULES, INSERT CODE, AND RUN YOUR PROGRAMS HERE
```

Lesson 3.4:
Slurm Job Script Generation

- ITS has set up a wonderful tool for automatically generating Slurm scripts

https://www.hpc.iastate.edu/guides/condo-2017/slurm-job-script-generator-for-condo

- We will be able to use Slurm on our training server; you can get access to condo through your lab:

https://www.hpc.iastate.edu/guides/condo-2017/who-can-use-condo

Lesson 3.4:
Slurm Job Script Generation

Slurm Job Script Generator for Condo

* indicates a required field

Parameters	
* Compute node type: *Users in Free Tier select freecompute*	compute ▼
* Number of compute nodes: *Use 1 unless running a distributed-memory parallel application*	1
* Number of processor cores per node: *regular compute nodes have 16 cores, free compute nodes have 12 cores* *fat node has 32 cores, huge node has 40 cores, GPU node has 12 cores*	1
* Walltime limit: *Maximum time the job may run. Shorter jobs may start running sooner.*	1 hours 00 mins 00 secs
Max memory per compute node: *Default memory is 8000 MB per processor core requested for the regular compute nodes (1 GB == 1024 MB)*	64 MB ▼
Job name: *Default is the job script name. Displayed by the squeue command.*	testJob
Receive email for job events: *BEGIN: when job starts, END: when job ends, FAIL: if job fails*	☐ BEGIN ☐ END ☐ FAIL my.email@iastate.edu EMAIL
Job standard output file: *Default is "slurm-%j.out", where %j is the job ID.*	testJob.output
Job standard error file: *Default is same file as the job standard output file.*	testJob.error

Lesson 3.4:
Slurm Job Script Generation

Compute node type:

Allows you to select the node type for your job

You will mostly use "compute" or "freecompute"; gpu, fat, and huge are more expensive and for processing truly ridiculous amounts of information (like TB, as opposed to GB)

Parameters

* Compute node type:
Users in Free Tier select freecompute
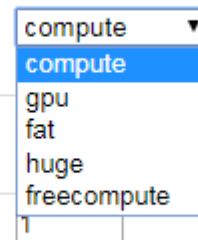
compute ▼
compute
gpu
fat
huge
freecompute

* Number of compute nodes:
Use 1 unless running a distributed-memory parallel application

* Number of processor cores per node:
regular compute nodes have 16 cores, free compute nodes have 12 cores

1

Lesson 3.4:
Slurm Job Script Generation

Number of compute nodes:

Allows you to select how many nodes will process your job

More nodes = faster completion = more $

* Number of compute nodes: ___1___
Use 1 unless running a distributed-memory parallel application

Lesson 3.4:
Slurm Job Script Generation

Number of processor cores per node:

Allows you to select how many processors in each node will be utilized

More processors = faster completion = more $

* Number of processor cores per node: 1

regular compute nodes have 16 cores, free compute nodes
have 12 cores

fat node has 32 cores, huge node has 40 cores, GPU node has
12 cores

Lesson 3.4:
Slurm Job Script Generation

Walltime limit:

Limits how long a job will run

A job will immediately be canceled if it runs into its walltime limit

Shorter walltimes tend to be prioritized (to finish as many jobs as quickly as possible)

* Walltime limit: 1   hours 00   mins 00   secs
Maximum time the job may run. Shorter jobs may start running sooner.

Lesson 3.4:
Slurm Job Script Generation

- Pro-tip (Thanks, Sagnik!):

If you're running a script for the first time, allocate 1-2min of walltime, instead of several hours or days

Check to make sure your script actually runs before requesting more than a few minutes of HPC time!

Lesson 3.4: Slurm Job Script Generation

Max memory per compute node:

Limits how much memory each node you are utilizing will use

More memory = faster completion = more $

Max memory per compute node: _____ MB ▾
Default memory is 8000 MB per processor core requested for the regular compute nodes (1 GB == 1024 MB)

Lesson 3.4:
Slurm Job Script Generation

Job name:

A name you can provide to more easily identify your jobs

Job name:
Default is the job script name. Displayed by the squeue command.

testJob

Lesson 3.4: Slurm Job Script Generation

Receive email for job events:

You can provide an email address to optionally receive emails to inform you when your job starts, when it ends, and if it fails due to an error of some kind

Receive email for job events:
BEGIN: when job starts, END: when job ends, FAIL: if job fails

☐ BEGIN ☐ END ☐ FAIL

my.email@iastate.edu

EMAIL

Lesson 3.4:
Slurm Job Script Generation

Job standard output file:

If your job would output information to the console, this is the name of the file it will output to

Job standard output file:
Default is "slurm-%j.out", where %j is the job ID.

testJob.output

Lesson 3.4:
Slurm Job Script Generation

Job standard error file:

If your job fails due to an error, the error message/log will be output to this file

Job standard error file:
Default is same file as the job standard output file.

testJob.error

Lesson 3.4:
Slurm Job Script Generation

As you specify options, the page will dynamically generate a script header containing your parameters

Job Script

```bash
#!/bin/bash

# Copy/paste this job script into a text file and submit with the command:
#    sbatch thefilename

#SBATCH --time=1:00:00   # walltime limit (HH:MM:SS)
#SBATCH --nodes=1   # number of nodes
#SBATCH --ntasks-per-node=1   # 1 processor core(s) per node
#SBATCH --mem=64M   # maximum memory per node
#SBATCH --job-name="testJob"
#SBATCH --output="testJob.output" # job standard output file (%j replaced by job id)
#SBATCH --error="testJob.error" # job standard error file (%j replaced by job id)

# LOAD MODULES, INSERT CODE, AND RUN YOUR PROGRAMS HERE
```

Below this section, you can write code to execute some function/run a program

Lesson 3.4:
Slurm Job Script Generation

When creating Slurm commands, you can:

1. Open your script file with `nano`

2. Copy the header script to your clipboard (**[Ctrl]** + **[c]** or **[command]** + **[c]**)

3. Then **Left click** or **[command]** + **click** to paste the header directly into the console

Lesson 3.4:
Slurm Job Script Generation

- We are not on condo
- We are on a training server
- Use these settings for Slurm jobs on training server:

Parameters

* Compute node type: compute ▾
Users in Free Tier select freecompute

* Number of compute nodes: 1
Use 1 unless running a distributed-memory parallel application

* Number of processor cores per node: 1
*regular compute nodes have 16 cores, free compute nodes
have 12 cores
fat node has 32 cores, huge node has 40 cores, GPU node has
12 cores*

* Walltime limit: 0   hours 5   mins 00   secs
*Maximum time the job may run. Shorter jobs may start running
sooner.*

Max memory per compute node: 0.1   MB ▾
*Default memory is 8000 MB per processor core requested for
the regular compute nodes (1 GB == 1024 MB)*

Exercise 3 Prelude

- Use https://www.hpc.iastate.edu/guides/condo-2017/slurm-job-script-generator-for-condo to generate Slurm headers for your file

- Name a standard output file and a standard error

- Copy the Job Script

- Create a copy of the file called `slurm-test.sh`; call this copy `<yourname>-slurm-test.sh`

- Open `<yourname>-slurm-test.sh` and paste your Slurm Job Script header above the code in `<yourname>-slurm-test.sh`

- Change "0.1M" to "1K"

Lesson 3.5: Slurm Commands

3.51: Submit a Slurm script

3.52: Check the job queue

3.53: Cancel a Slurm job
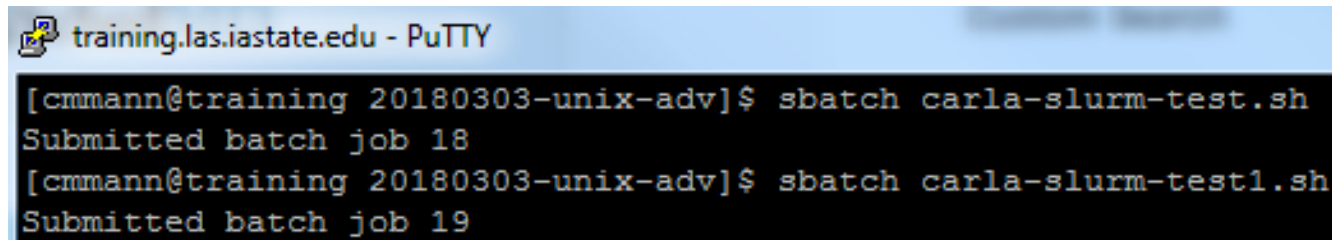
3.54: Check job stats

Lesson 3.51: Submit a Slurm script

Command: `sbatch <slurmscript.sh>`

What it does:

sbatch submits a script to Slurm, which reads the parameters in the #SBATCH lines and then allocates resources and distributes jobs accordingly


```
training.las.iastate.edu - PuTTY
[cmmann@training 20180303-unix-adv]$ sbatch carla-slurm-test.sh
Submitted batch job 18
[cmmann@training 20180303-unix-adv]$ sbatch carla-slurm-test1.sh
Submitted batch job 19
```

Your job will be assigned a job ID; you can use this job ID to check job stats, cancel jobs, etc.

Lesson 3.52: Check the job queue

Command: `squeue -options`

What it does:

View information on jobs in the Slurm queue

Options:

`-u <user>`: View information only on jobs submitted by a particular user

```
[cmmann@training 20180303-unix-adv]$ squeue
         JOBID  PARTITION     NAME      USER ST       TIME   NODES NODELIST(REASON)
            17   training carlaSlu   cmmann PD       0:00       1 (Resources)
            16   training carlaSlu   cmmann  R       0:56       1 localhost
```

Lesson 3.53: Cancel a Slurm job

Command: `scancel <jobid>`

What it does:

Immediately stops running `<jobid>`, or removes `<jobid>` from the Slurm queue

```
[cmmann@training 20180303-unix-adv]$ squeue
        JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           21  training carlaSlu   cmmann PD       0:00      1 (Resources)
           20  training carlaSlu   cmmann  R       0:06      1 localhost
[cmmann@training 20180303-unix-adv]$ scancel 21
[cmmann@training 20180303-unix-adv]$ squeue
        JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           20  training carlaSlu   cmmann  R       0:14      1 localhost
```

Exercise 3

1. Submit `<yourname>-slurm-test.sh` to Slurm for processing.
The code should take exactly 30 seconds to run, once it reaches the head of the queue.

2. Use `squeue` to view the Slurm queue

3. Try to cancel SOMEONE ELSE's job. What happens?

4. View your output file. What did the code in `<yourname>-slurm-test.sh` do?

Lesson 4:

Overview:

Lesson 4.0: Text Output

Commands:

```
cat <filename.txt>
head <filename.txt>
tail <filename.txt>
less <filename.txt>
```

What they do:

`cat` outputs the entirety of `<filename.txt>` to the console (don't try this with large files!!)

`head` outputs the first 10 lines of the file

`tail` outputs the last 10 lines of the file

`less` lets you scroll around a file

Lesson 4.1: Word Count
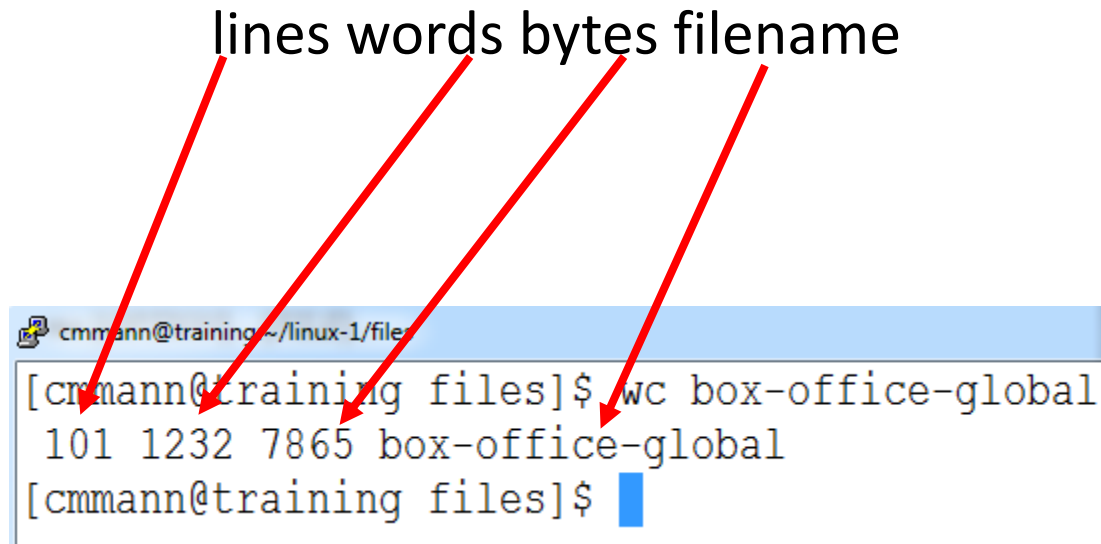
Command:

`wc <filename>`

What it does:

Outputs the number of:

lines words bytes filename

```
cmmann@training ~/linux-1/file
[cmmann@training files]$ wc box-office-global
  101 1232 7865 box-office-global
[cmmann@training files]$
```

Lesson 4.1: Word Count

Options:

`-l` : output ONLY the number of LINES and filename

`-w` : output ONLY the number of WORDS and filename

`-m` : print the number of characters in the file and filename

Lesson 4.1: Word Count

So that's cool, but these options all put out the filename as well

How do we get around that?

Many, many possible ways, but we're going to use piping for now

Lesson 4.2: Piping

We can take the output of one command, and directly feed it to another command – all in one line, using the **[|]** key (this is generally directly below the backspace key)

```
command1 | command2
```

Example:

```
[cmmann@training files]$ wc bill-of-rights
  10  482 2797 bill-of-rights
[cmmann@training files]$ cat bill-of-rights | wc -l
10
```

Lesson 4.3: sort

Command:

`sort <file>`

What it does:

Sorts `<file>` alphabetically by line

Options:

`-n` : sort numerically (if there are no numbers, it will default to alphabetic sort

`-r` : sort in reverse alphabetical order

`-u` : sort only unique items

Lesson 4.4: uniq

Command:
`uniq`

What it does:
Finds unique occurrences of text input

Options:
`-c` : count the occurrences of each line
`-d` : print only duplicated lines
`-u` : print only unique lines

Lesson 4.4: uniq

`uniq` must be called on something that is already sorted!

It works by comparing adjacent items in a list and discarding if they are identical.

Generally called after sort:

```
cat hello.txt | sort | uniq
```

Lesson 4.5: Grep

Stands for "Global Regular Expression Print"

EXTREMELY POWERFUL search tool

Finds text matching highly variable criteria and prints the lines containing that text

Can search multiple files, and find the files that match
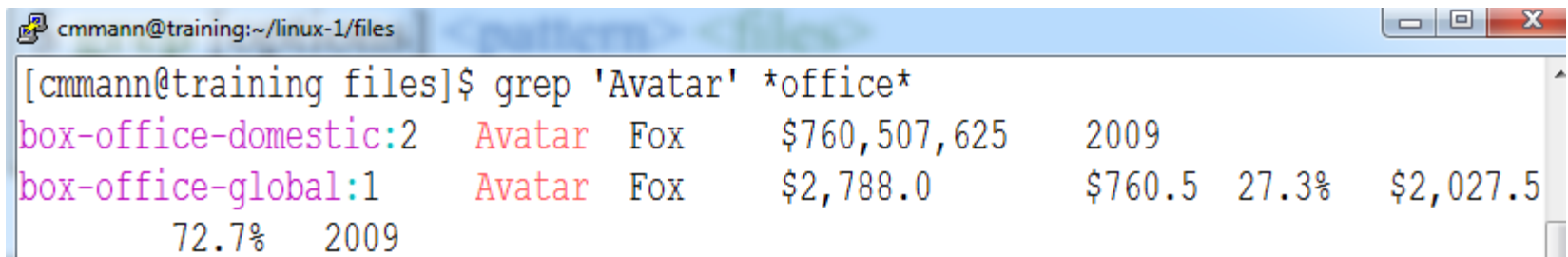
Lesson 4.5: Grep

Command:

```
grep -options <pattern> <files>
```

What it does:

     `grep` searches `<files>` for content matching `<pattern>`

Lesson 4.6: Piping and Grep

We can also feed text directly to grep, and have it search that:

Command:

`<intext> | grep -options <pattern>`

What it does:

grep searches `<intext>` for content matching `<pattern>`

```
[cmmann@training files]$ cat box-office-global | grep 'Avatar'
1       Avatar  Fox     $2,788.0        $760.5  27.3%   $2,027.5        72.7%
009
```
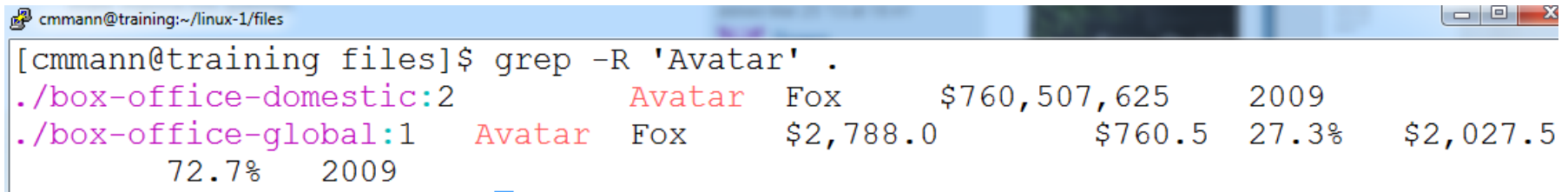
Lesson 4.7:
Grep All Files in A Directory

We can also search for content within a directory:

```
grep -R <pattern> <directoryname/>
```

For this, we have to use the `-R` Recursive option!


```
cmmann@training:~/linux-1/files
[cmmann@training files]$ grep -R 'Avatar' .
./box-office-domestic:2          Avatar   Fox      $760,507,625     2009
./box-office-global:1    Avatar  Fox      $2,788.0          $760.5   27.3%    $2,027.5
         72.7%    2009
```

Lesson 4.8: Grep Options

Grep has many, many options:

`-c` : count how many LINES on which the pattern occurs

`-o` : show only the part of a line that matches a pattern; this will show all matches in the line

`-v` : invert match – so select things that DON'T match `<pattern>`

`-i` : case insensitive matching

`-l` : list the files with a match

`-L` : list the files that don't have a match

Exercise 4: Hello

1. Navigate to `~/20180303-adv-unix/exercise4/`

2. Open "`exercise4.sh`"

3. Edit the file to perform the exercises.

4. Execute the file!

Hint:

      If you aren't sure if you're getting the correct answers, you can run `exercise4answers.sh`.

Lesson 5: Regular Expressions

Overview:

Lesson 5.1: Regular Expressions

Also called 'regex' or 'regexp'

UNBELIEVABLY POWERFUL tool for defining search patterns

Consists of 'codes' that denote various conditions

These conditions can be used to very narrowly find things, or very, very broadly find things

Lesson 5.2: Regular Expressions

In UNIX, frequently used with `grep`

```
grep -E <'regexppattern'> <file>
```

The option "`-E`" tells grep that the pattern is a regular expression!

It is very important that you remember the "-E" option, otherwise grep will try to match your exact pattern, instead of what it represents.

Lesson 5.2: egrep

Alternatively, you can use egrep:

```
egrep <'regexppattern'> <file>
```

This behaves exactly as "`grep -E`", and will be used through the rest of the slides.

Lesson 5.3: Matching Words

We can still match words while grepping regular expressions:

```
egrep 'cat' <file>
```

will still find any instance of the letters 'cat' in a file

But grep allows us to search for words
similar to 'cat'…

Lesson 5.4: Fuzzy Matching

The regexp to find words containing 'cat' or 'cot' would be:
'[c][ao][t]'

The brackets encase 'character' slots

What would '[fw][i][s][h]' match?

Lesson 5.4: Fuzzy Matching

The regexp to find words containing 'cat' or 'cot' would be:

'[c][ao][t]'

The brackets encase 'character' slots

What would '[fw][i][s][h]' match?

Lesson 5.4: Fuzzy Matching

If we only wanted to match the 'word' cat or cot, and not , we can bracket '[c][ao][t]' with spaces:

'[ ][c][ao][t][ ]'

Note that many, many systems use Regular Expressions, and some have slightly different usage.

For many systems, you can specify a match to 'whitespace' (spaces and tabs) using "\s", but this does not work in bash.

Lesson 5.4: Fuzzy Matching

This bracket system, though, is rather cumbersome. Instead, we could specify:

'\b[c][ao][t]'

In this context, '\b' means to match the beginning of a word.

The '\' before the 'b' is an *escape* character – it signals that we don't want to *literally* match the letter 'b', but the condition that 'b' represents.

Lesson 5.4: Fuzzy Matching

We can use a '-' to represent a span of characters:

'[a-c][o][g]' would recognize 'aog', 'bog', 'cog'

'[l-z][o][g]' would only recognize 'log', 'mog', 'nog' etc.

Lesson 5.5: Number Matching

We can also match numbers:

What number(s) will the following match?

'[0-3][5-8][345]'

Lesson 5.5: Number Matching

We can also match numbers:

What number(s) will the following match?

'[0-3][5-8][345]'

Lesson 5.6: Operators

'[cd][ao][tg]' would match 'cat' or 'dog'

(But also 'cag', 'dat', and any combination of those letters)

What regular expression would you use to find words containing "trap" or "tarp"?

Lesson 5.6: Operators

'[cd][ao][tg]' would match 'cat' or 'dog'

(But also 'cag', 'dat', and any combination of those letters)

What regular expression would you use to find words containing "trap" or "tarp"?

Lesson 5.6: Operators

But what if we wanted to match 'trap' or 'tarp', but not 'trrp' or 'taap'?

We can use *operators* to specify this!

Lesson 5.6: Operators

If you want to match *this* OR *that*:

```
egrep 'this|that' <file>
```

When using regular expressions, grep understands that "|" means "OR"

If you to find things that are NOT something, you use:

```
egrep -v 'something' <file>
```

Lesson 5.6: Operators

What if you want to match the character '|'?

We use escape characters again!

```
egrep '\|'
```

Lesson 5.7:
Matching X Letters

What if we want to find something more complicated, like a zip code?

What is the form of a zip code?

How could we potentially match that?

Lesson 5.7:
Matching X Letters

But that's rather cumbersome. Instead, we can specify a specific number of times to look for a set of characters:

```
egrep '[0-9]{5}' <file>
```

In regexp, you can use a number in brackets AFTER the thing that you want to repeat

Lesson 5.7:
Matching X Letters

We can put more than just a number in there:

`a{n,}` : will match the letter 'a' `n` OR MORE times

What will 'a{2,}' match?

aardvark, armadillo, aaaah

Lesson 5.7:
Matching X Letters

We can put more than just a number in there:

`a{n,}` : will match the letter 'a' `n` OR MORE times

What will '`a{2,}`' match?

aardvark, armadillo, aaaah

Lesson 5.7: Matching X Letters

We can also specify a range of times to match:

`a{n, m}` : will match 'a' at least `n` times, but not more than `m` times.

What will '`a{2, 3}`' match?

aardvark, armadillo, aaaah

Lesson 5.7: Matching X Letters

We can also specify a range of times to match:

`a{n, m}` : will match 'a' at least `n` times, but not more than `m` times.

What will '`a{2, 3}`' match?

aardvark, armadillo, aaaah

Lesson 5.7:
Matching X Letters

We can also specify more matches:

`a*` : match 'a' 0 or more times

`a+` : match 'a' 1 or more times

`a?` : match 'a' once if it happens, but matching it is optional

Lesson 5.8: Example

We can match EXTREMELY complicated things

Real world example: PDB files

In my day job, I want to find the coordinates of atoms in PDB files.

These lines take the form:

```
ATOM        1   N    SER A   44       0.312   28.338   23.824   1.00109.80              N
ATOM        2   CA   SER A   44      -1.014   28.655   23.237   1.00113.84              C
ATOM        3   C    SER A   44      -1.893   27.385   23.044   1.00115.10              C
ATOM        4   O    SER A   44      -1.573   26.307   23.566   1.00111.94              O

ATOM     1589   O3'    A B    9       4.770   39.279   56.136   1.00228.34              O
ATOM     1590   C2'    A B    9       2.693   40.521   56.600   1.00214.10              C
ATOM     1591   O2'    A B    9       3.406   41.227   57.593   1.00219.27              O
ATOM     1592   C1'    A B    9       1.906   41.493   55.715   1.00207.15              C
```

Lesson 5.8: Example

So these lines look similarly, but they have different numbers and characters spaced differently.

And the rest of the file looks NOTHING like this.

How could I pull out ONLY these lines?

```
ATOM        1  N    SER A  44        0.312   28.338   23.824   1.00109.80              N
ATOM        2  CA   SER A  44       -1.014   28.655   23.237   1.00113.84              C
ATOM        3  C    SER A  44       -1.893   27.385   23.044   1.00115.10              C
ATOM        4  O    SER A  44       -1.573   26.307   23.566   1.00111.94              O

ATOM     1589  O3'   A B   9        4.770   39.279   56.136   1.00228.34              O
ATOM     1590  C2'   A B   9        2.693   40.521   56.600   1.00214.10              C
ATOM     1591  O2'   A B   9        3.406   41.227   57.593   1.00219.27              O
ATOM     1592  C1'   A B   9        1.906   41.493   55.715   1.00207.15              C
```

Lesson 5.8: Example

We could try:

`egrep 'ATOM' 1R2X.pdb,` but...

```
REMARK 290
REMARK 290 CRYSTALLOGRAPHIC SYMMETRY TRANSFORMATIONS
REMARK 290 THE FOLLOWING TRANSFORMATIONS OPERATE ON THE ATOM HETATM
REMARK 290 RECORDS IN THIS ENTRY TO PRODUCE CRYSTALLOGRAPHICALLY
```

```
ATOM      1  N    SER A  44        0.312  28.338  23.824  1.00109.80           N
ATOM      2  CA   SER A  44       -1.014  28.655  23.237  1.00113.84           C
ATOM      3  C    SER A  44       -1.893  27.385  23.044  1.00115.10           C
ATOM      4  O    SER A  44       -1.573  26.307  23.566  1.00111.94           O

ATOM   1589  O3'   A B   9        4.770  39.279  56.136  1.00228.34           O
ATOM   1590  C2'   A B   9        2.693  40.521  56.600  1.00214.10           C
ATOM   1591  O2'   A B   9        3.406  41.227  57.593  1.00219.27           O
ATOM   1592  C1'   A B   9        1.906  41.493  55.715  1.00207.15           C
```

Lesson 5.8: Example

We can specify that we only want to match 'ATOM' if it starts at the beginning of the line:

```
egrep '^ATOM' 1R2X.pdb
```

The character '^' is a special character called an 'anchor' that means to match the beginning of the line

Lesson 5.8: Example

But what if I ONLY want the protein atom coordinates?


```
ATOM       1  N    SER A   44         0.312   28.338   23.824   1.00109.80              N
ATOM       2  CA   SER A   44        -1.014   28.655   23.237   1.00113.84              C
ATOM       3  C    SER A   44        -1.893   27.385   23.044   1.00115.10              C
ATOM       4  O    SER A   44        -1.573   26.307   23.566   1.00111.94              O
```

```
ATOM    1589  O3'   A  B    9         4.770   39.279   56.136   1.00228.34              O
ATOM    1590  C2'   A  B    9         2.693   40.521   56.600   1.00214.10              C
ATOM    1591  O2'   A  B    9         3.406   41.227   57.593   1.00219.27              O
ATOM    1592  C1'   A  B    9         1.906   41.493   55.715   1.00207.15              C
```

Lesson 5.8: Example

We make a really *complicated* regexp:

```
egrep '^ATOM[ ]*[0-9]+[ ]*[A-Z]+[
]*[A-Z]{3}'
```

```
ATOM      1  N    SER A  44      0.312  28.338  23.824  1.00109.80      N
ATOM      2  CA   SER A  44     -1.014  28.655  23.237  1.00113.84      C
ATOM      3  C    SER A  44     -1.893  27.385  23.044  1.00115.10      C
ATOM      4  O    SER A  44     -1.573  26.307  23.566  1.00111.94      O
```

Lesson 5.8: Example

What is this doing?

Match ATOM at the beginning of the line

Match at least one letter

Match at least one digit

`` `^ATOM[ ]*[0-9]+[ ]*[A-Z]+[ ]* ``

Match any number of spaces

`[A-Z]{3}'`

Match exactly 3 letters

Lesson 5.9:
Regexp Continuing Education

There are many, many more options available to use with regexp in bash

We could spend an entire workshop on this alone. (We're not going to today, though.)

If you want to learn more, visit:

http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_04_01.html#sect_04_01_02

Exercise 5: Real Life Stuff

1. Open and edit `exercise5.sh`

2. Complete the exercises within.

3. Run `exercise5.sh`

Closing

Thanks for coming!

Please take this survey so that we can improve the workshop for future attendees:

https://goo.gl/forms/cumiG8DvCOzOmDuC2