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 on “Statistics (SAS)”
- Hit [Ctrl]+[Alt]+[Del]
- Double click the desktop folder “SSH & Secure File Transfer”
- Double click on “putty.exe”
- Sign into the server
- Chill

Advanced UNIX

BCBGSO Workshop

March 2nd, 2017

Presenter: Carla Mann

Thanks!

- BIG thanks to Jennifer Chang for inspiration for slides and materials
- Organizers: Ashish Jain and Dan Kool
- Funding/Support/Volunteers: BCBGSO
- Tech support: Biology IT (Levi Baber)

Our many, MANY volunteers:

Dan Kool
Sagnik Banerjee
Lindsay Rutter
Nancy Machanda
John Hsieh

Ashish Jain
Akshay Yadav
Alvin Chon
Cam Fay
Ashley Zhu

David Hufnagel
Schuyler Smith
Avani Khadilkar
Sayane Shome
Gaurav Kandoi

Xiyu Peng
Urminder Singh
Kumar Ambuj
Rebekah Starks
Michael Zeller

Materials

All materials (slides, cheat sheets, etc) available here:

https://github.com/cmmann/advanced-unix-workshop-materials.git

Set-Up

Mac/Linux:

Open terminal

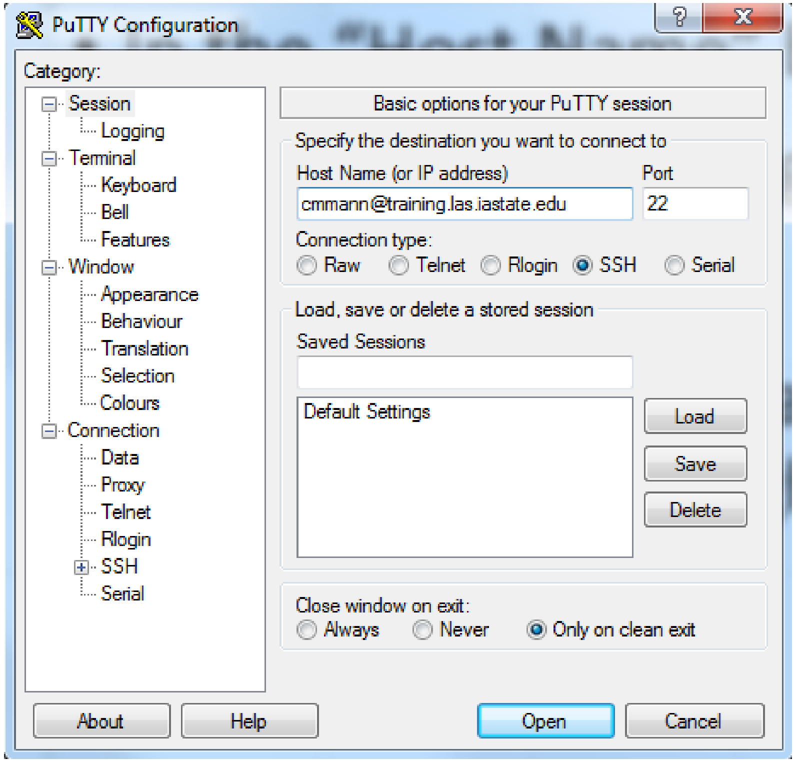
```
ssh <your-netid>@training.las.iastate.edu
```

Windows:

Open Putty.exe

Enter

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



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): cmmann@training.las.iastate.edu

Port: 22

Connection type: ○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Load, save or delete a stored session

Saved Sessions: Default Settings

Close window on exit: ○ Always ○ Never ● Only on clean exit

Overview

Lesson 0: Quick Review of Basic UNIX

Lesson 0.5: Setup

Lesson 1: Text Editing with nano

Lesson 2: Shell Scripting; if and loop

Lesson 3: Data Exploration with `wc` and `grep`

Lesson 4: Regular Expressions

Lesson 5: Further Data Exploration with `sort` and `uniq`

Lesson 5: awk

Lesson 0: Quick Review

In PowerPoint, commands you will type will look
`like this`

`cd`: change directory

`ls`: list directory contents

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

Lesson 0.5: Setup

Once logged in:

```
git clone https://github.com/cmmann/20170302-adv-unix.git
```

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" is `nano`

*If you already know how to use Vim, go ahead and use it!

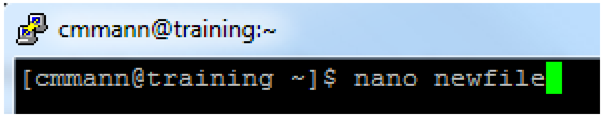
Lesson 1.2: Text Editing with nano

To use:

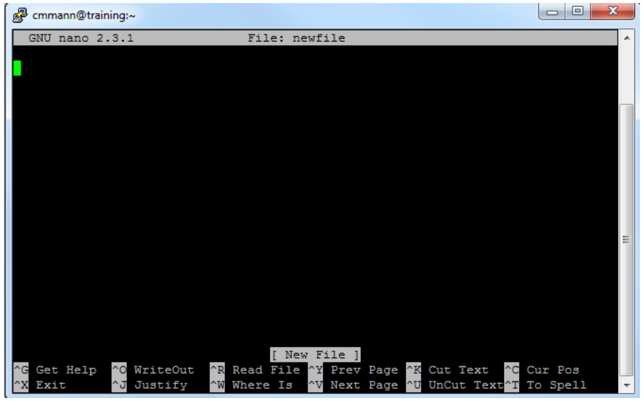
```
nano <filename>
```

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



```
[cmmann@training ~]$ nano newfile
```



```
  GNU nano 2.3.1                File: newfile

                               [ 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
```

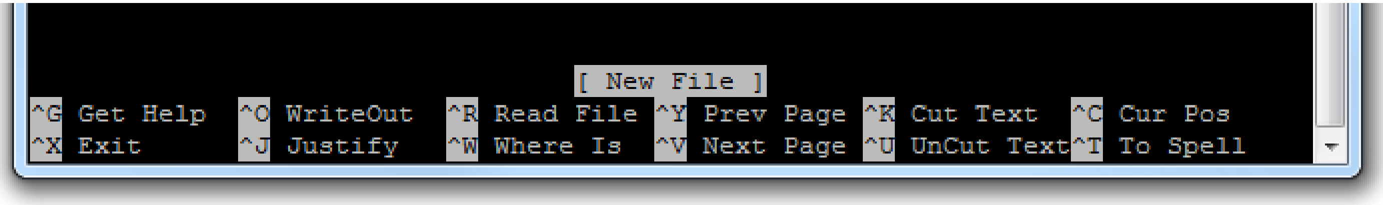
Lesson 1.2: Text Editing with nano

`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 or another 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
   ```
   cd 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
   ```
   cd adv-unix/exercise1
   ```
2. Use nano to open “exercise1.txt”
   ```
   nano 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

   ```
   cd adv-unix/exercise1
   ```

2. Use nano to open “exercise1.txt”

   ```
   nano 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)

   ```
   Carla
   I seek the Holy Grail
   Blue
   42
   ```

5. Exit (save when prompted)

Exercise 1:

Goals:

1. Navigate to adv-unix/exercise1
   ```
   cd adv-unix/exercise1
   ```
2. Use nano to open “exercise1.txt”
   ```
   nano 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)
   ```
   Carla
   I seek the Holy Grail
   Blue
   42
   ```
5. Exit (save when prompted)
   ```
   [Ctrl]+[x]
   [y]
   ```

Lesson 2: Shell Scripting

Overview:

Lesson 2.0: What is a shell?

Lesson 2.1: Creating a shell script

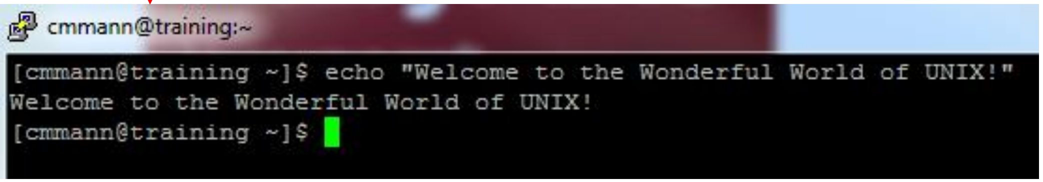
Lesson 2.3: Executing a shell script

Exercise 2: Completed throughout the lesson

Lesson 2.0: What is a shell?

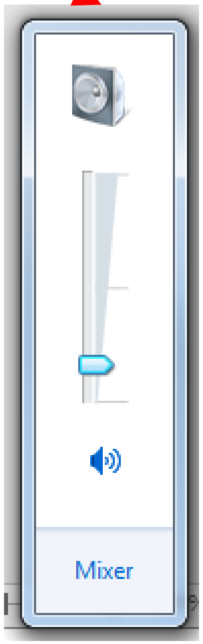
A shell is an interface for accessing an operating system’s services

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 ~]$ 
```



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:

$_ BASH

THE BOURNE-AGAIN SHELL

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 pasted 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

Create a file, using `nano`, called “`hello.sh`”

Lesson 2.1: Creating a Shell Script

WE know the script should be executed with bash

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
```

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

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

#Using a comment, add your name and the date to this script

Lesson 2.1: Comments

It is good practice to comment your scripts well

Will you still know what your script is doing 6 months from now?

Write extremely complicated code with no comments that works
Instant job security
quickmeme.com

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

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 current directory) by typing:

```
./commandname
```

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 /
```

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`?

```
./hello.sh
```

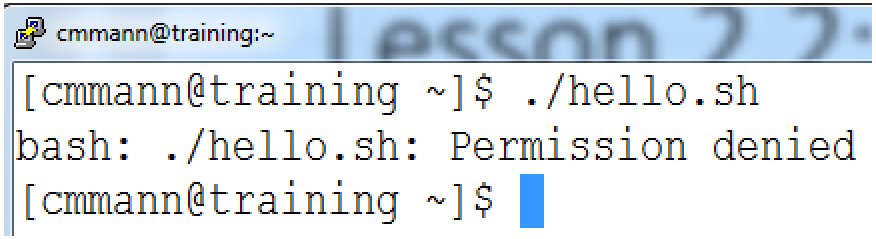
What happens when you run the script?

Lesson 2.2: Executing a Shell Script

So how do we execute `hello.sh`?

```
./hello.sh
```

What happens when you run the script?



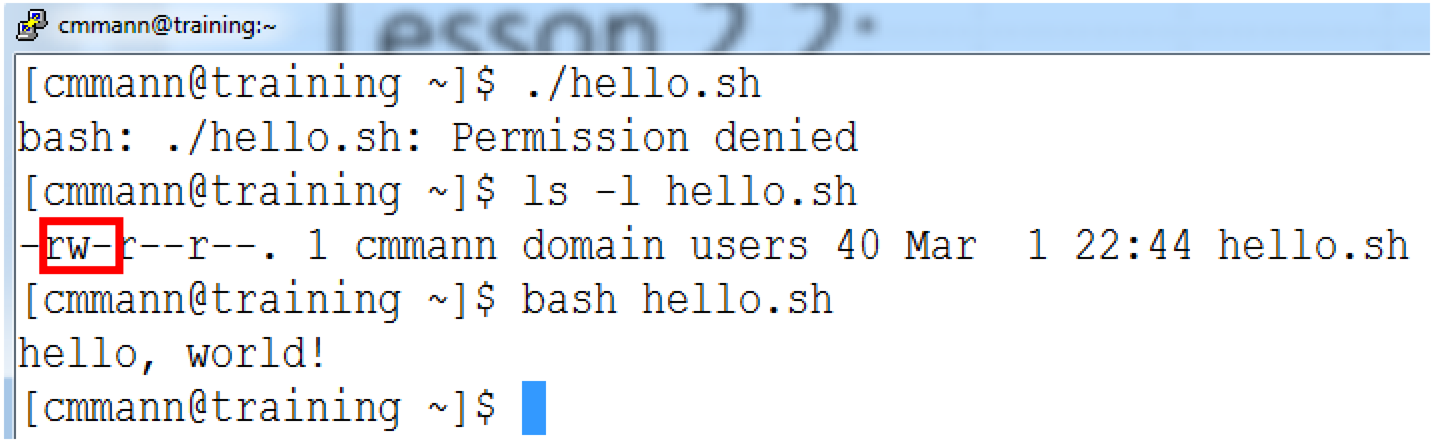
```
[cmmann@training ~]$ ./hello.sh
bash: ./hello.sh: Permission denied
[cmmann@training ~]$
```

Lesson 2.2: Executing a Shell Script

See what happens with:

```
bash hello.sh
```

Lesson 2.2:
Executing a Shell Script



```
[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

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 the commands in `hello.sh` and then executing those commands

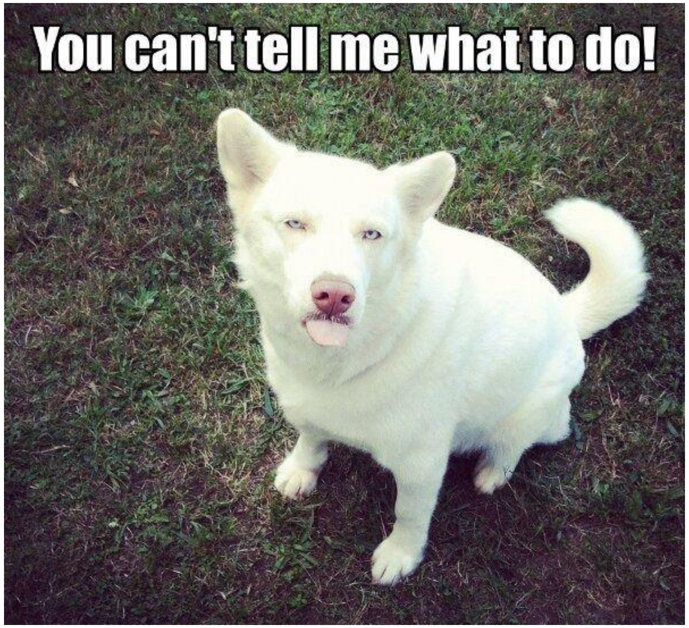
Lesson 2.3: Executing a Shell Script

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.



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

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

```
chmod 755 hello.sh
```

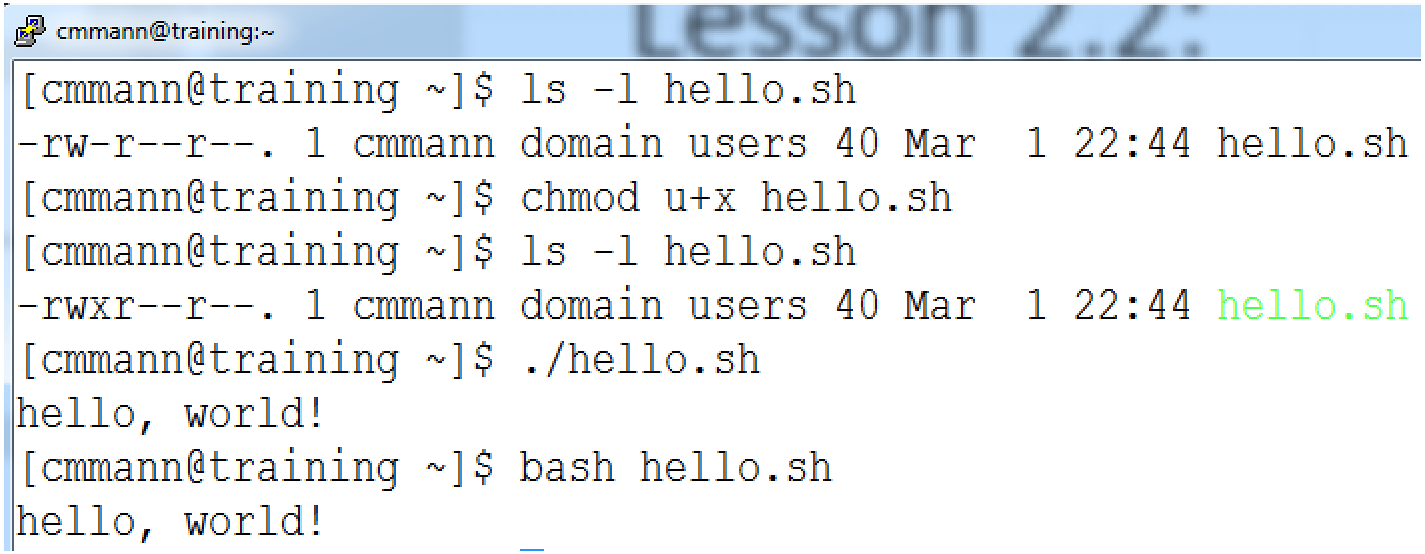
Alternate shortcut:

```
chmod u+x hello.sh
```

Lesson 2.3:
Executing a Shell Script

Try executing the script now:

```
/.hello.sh
```



```
[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

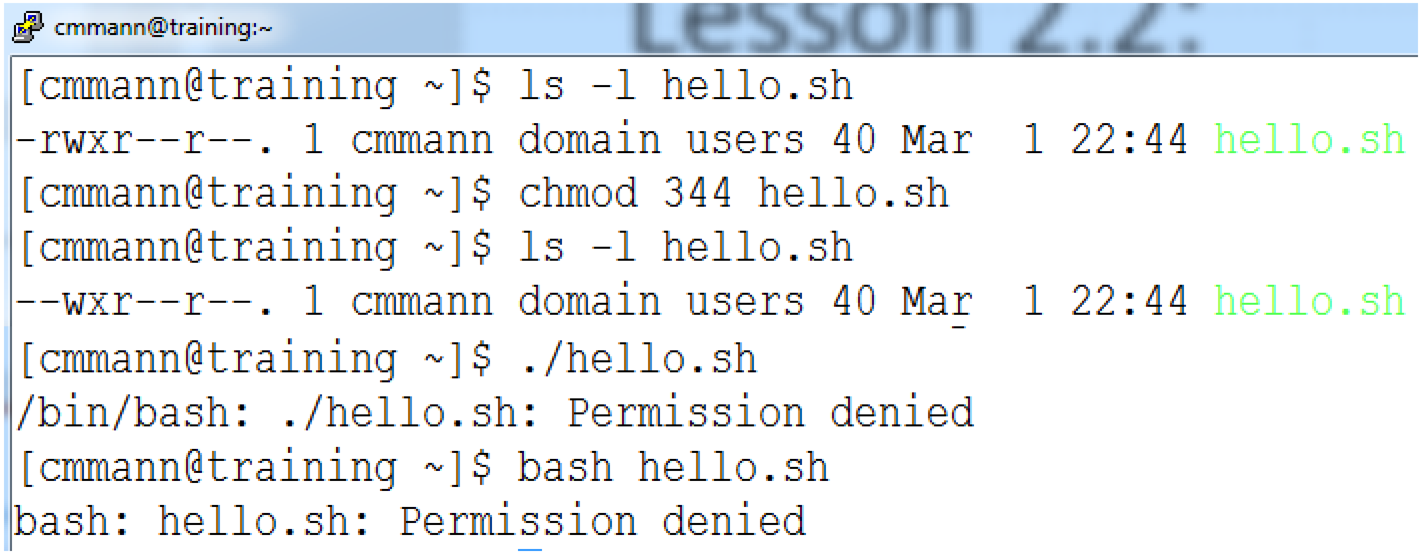
If execute permission isn't providing security, then what is?

Read and Write permissions!

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

```
chmod 344 hello.sh
```

Lesson 2.3:
Executing a Shell Script



```
[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:

Overview:

Lesson 3.0: Review Text Output

Lesson 3.1: Word Count

Lesson 3.2: Piping, overwriting, and appending

Lesson 3.3: Uniq

Lesson 3.4: Sort

Lesson 3.5: Grep

Exercise 3: Hello

Lesson 3.0: 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 3.1: Word Count

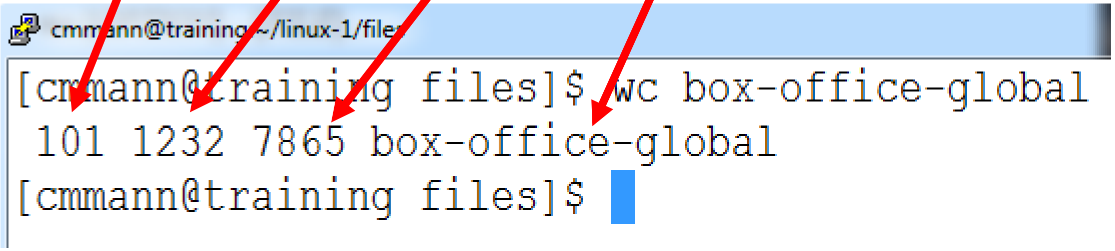
Command:

```
wc <filename>
```

What it does:

Outputs the number of:

lines words bytes filename



```
[cmmann@training files]$ wc box-office-global
 101 1232 7865 box-office-global
[cmmann@training files]$
```

Lesson 3.1: 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 3.1: 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 3.2: Piping

We can take the output of one command, and directly feed it to another command – all in one line, using the [|] key

```
command1 | command2
```

Example:

```
     cat bill-of-rights | wc -l
```

Lesson 3.3: 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 3.4: 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 3.4: 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 3.5: 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

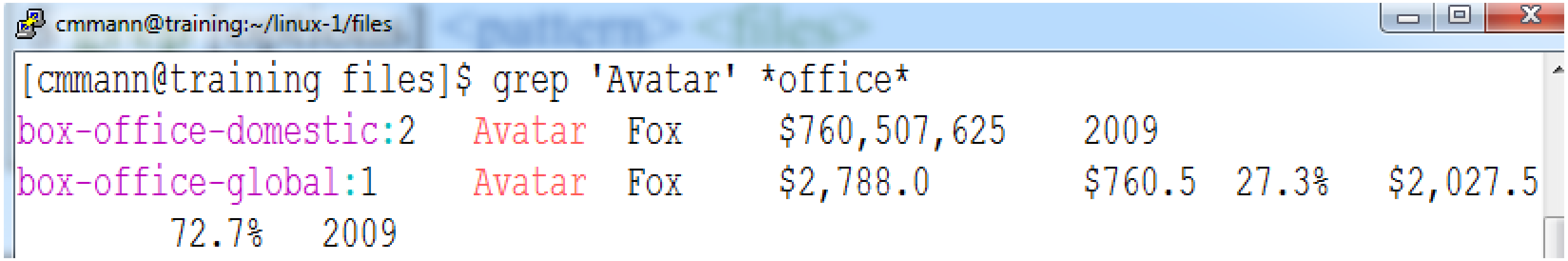
Lesson 3.5: Grep

Command:

`grep -options <pattern> <files>`

What it does:

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



```
[cmmann@training files]$ grep 'Avatar' *office*
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 3.6: Piping and Grep

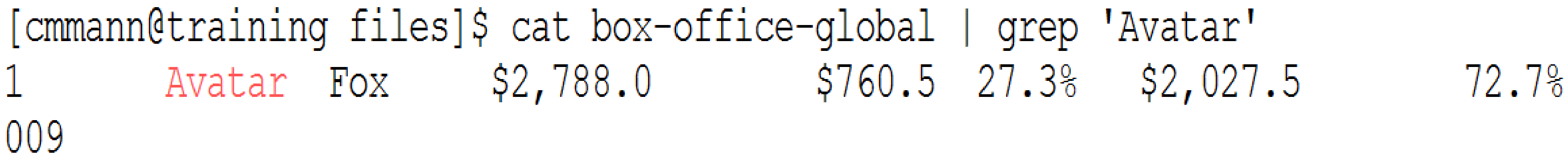
We can also feed text directly to grep, and have it search that:

Command:

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

What it does:

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



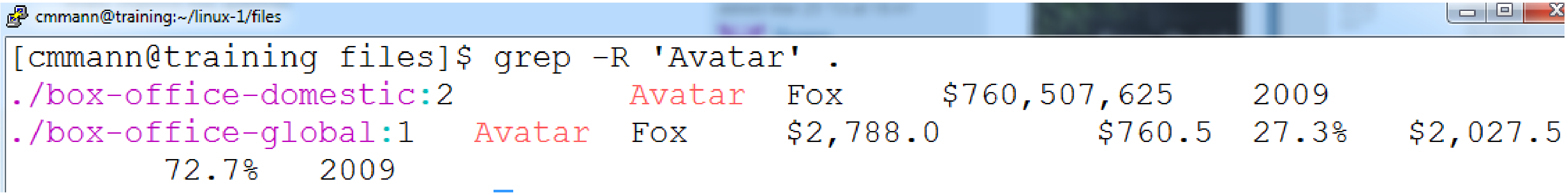
```
[cmmann@training files]$ cat box-office-global | grep 'Avatar'
1       Avatar  Fox     $2,788.0        $760.5  27.3%   $2,027.5        72.7%
009
```

Lesson 3.7:
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 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 3.8: 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 3: Hello

Navigate to `~/20170302-adv-unix/exercise3/`

Open “`exercise3.sh`”

Edit the file to perform the exercises.

Execute the file!

Lesson 4: Regular Expressions

Overview:

Lesson 4.1: What are Regular Expressions?

Lesson 4.2: `egrep`

Lesson 4.3: Matching words with `egrep`

Lesson 4.4: Fuzzy Matching

Lesson 4.5: Number Matching

Lesson 4.6: Operators

Lesson 4.7: Matching X Letters

Lesson 4.8: Example

Lesson 4.9: Continuing Education

Lesson 4.1: 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 4.2: 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 4.2: 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 4.3: 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 4.4: 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 4.4: 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?

fish, lungfish, fishing, wish, wishing, swish, etc.

Lesson 4.4: 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 4.4: 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 4.4: 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 4.5: Number Matching

We can also match numbers:

What number(s) will the following match?

‘[0-3][5-8][345]’

Lesson 4.5: Number Matching

We can also match numbers:

What number(s) will the following match?

‘[0-3][5-8][345]’

053, 153, 374, etc.

Lesson 4.6: 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 4.6: 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”?

‘[t][ar][ar][p]’

Lesson 4.6: Operators

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

Lesson 4.6: Operators

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

We can use *operators* to specify this!

Lesson 4.6: 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 4.6: Operators

What if you want to match the character ‘|’?

We use escape characters again!

```
egrep ‘\|’
```

Lesson 4.7: Matching X Letters

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

Lesson 4.7: Matching X Letters

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

What is the form of a zip code?

Lesson 4.7:
Matching X Letters

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

What is the form of a zip code?

5 numbers

Lesson 4.7:
Matching X Letters

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

What is the form of a zip code?

5 numbers

How could we potentially match that?

Lesson 4.7:
Matching X Letters

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

What is the form of a zip code?

5 numbers

How could we potentially match that?

‘[0-9][0-9][0-9][0-9][0-9]’

Lesson 4.7: 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 4.7: 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 4.7:
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 4.7: 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 4.7: 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 4.7: 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

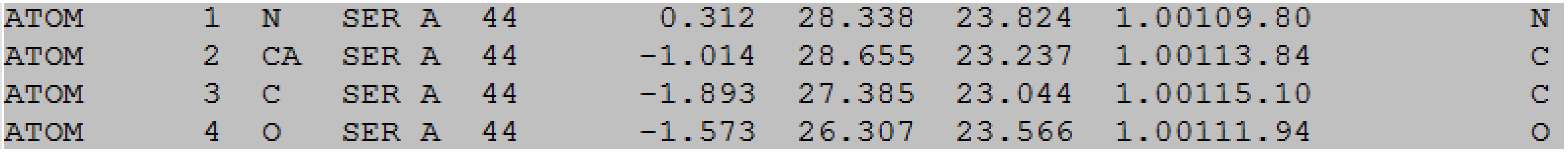
Lesson 4.8: Example

We can match EXTREMELY complicated things

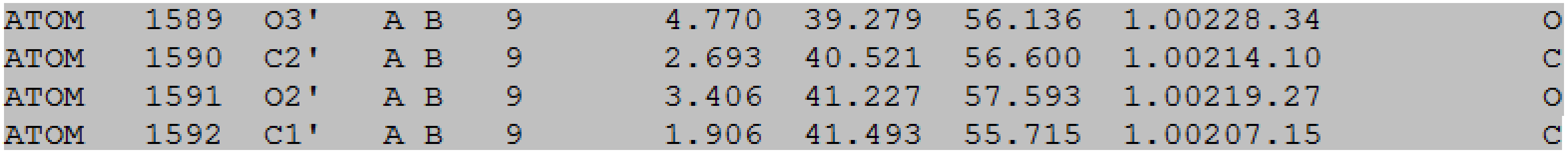
Real world example: PDB files

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
```



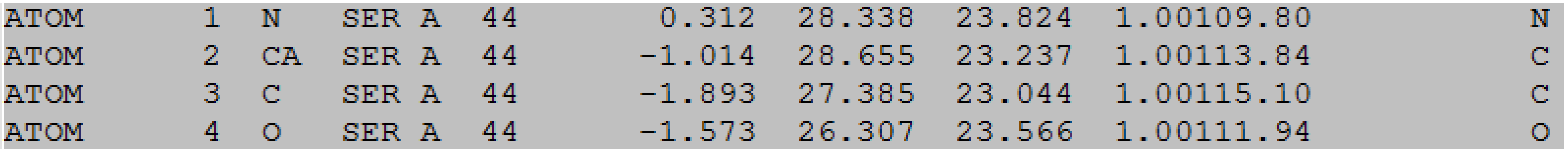
```
ATOM   1589  O3'   A B   9       4.770  39.279  56.136  1.00228.34           O
ATOM   1590  C2'   A B   9       2.693  40.521  56.600  1.00214.10           C
ATOM   1591  O2'   A B   9       3.406  41.227  57.593  1.00219.27           O
ATOM   1592  C1'   A B   9       1.906  41.493  55.715  1.00207.15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

Lesson 4.8: Example

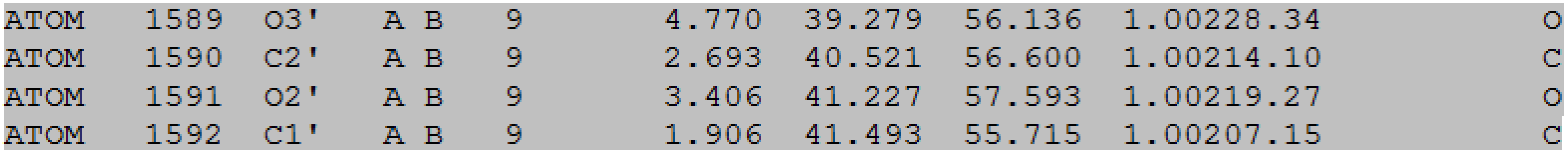
So these lines look similarly, but they have different numbers and characters spaced differently.

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
```

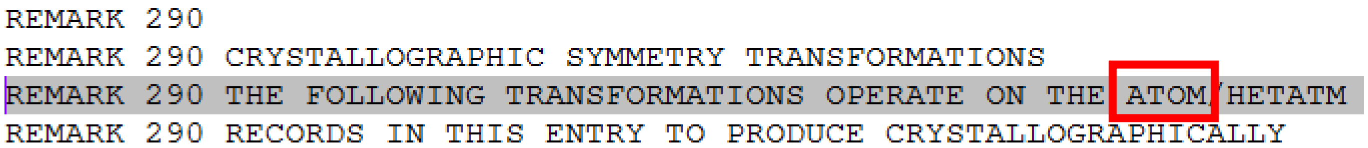


```
ATOM   1589  O3'   A B   9       4.770  39.279  56.136  1.00228.34           O
ATOM   1590  C2'   A B   9       2.693  40.521  56.600  1.00214.10           C
ATOM   1591  O2'   A B   9       3.406  41.227  57.593  1.00219.27           O
ATOM   1592  C1'   A B   9       1.906  41.493  55.715  1.00207.15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

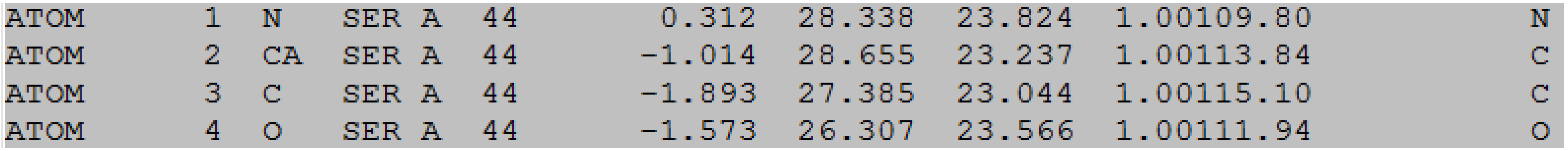
Lesson 4.8: Example

We could try:

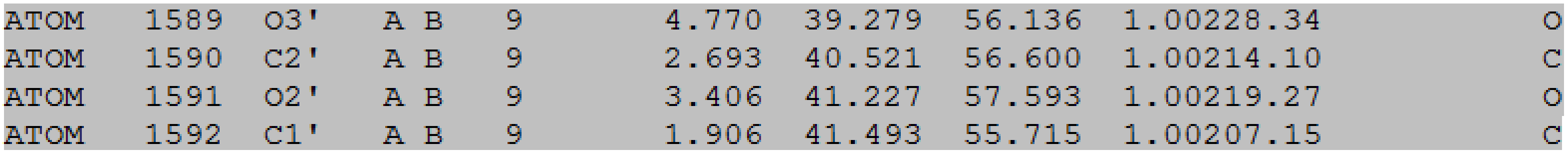
`egrep ‘ATOM’ 1R2X.pdb`, but…



```
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 4.8: Example

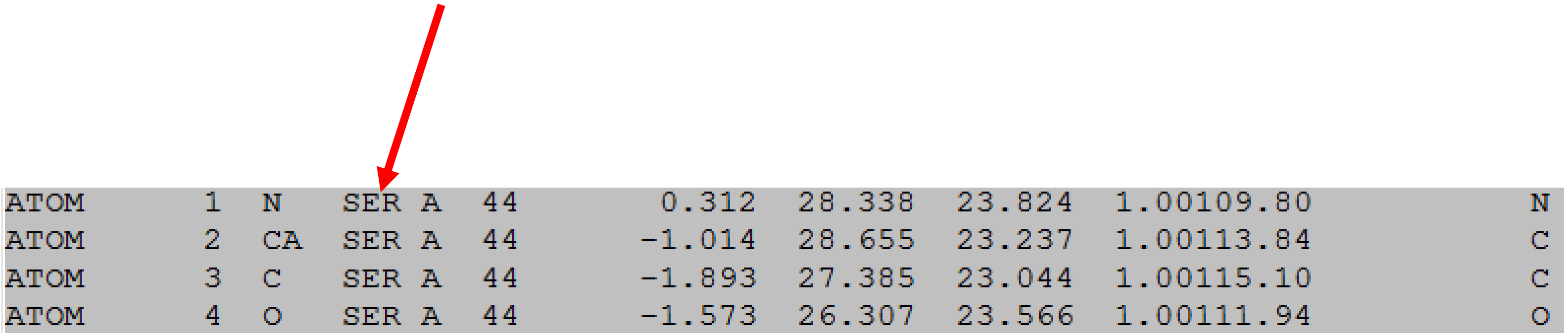
We can specify that we only want to match ‘ATOM’ if it starts at the beginning of the line:

```
egrep ‘^ATOM’ 1R2X.pdb
```

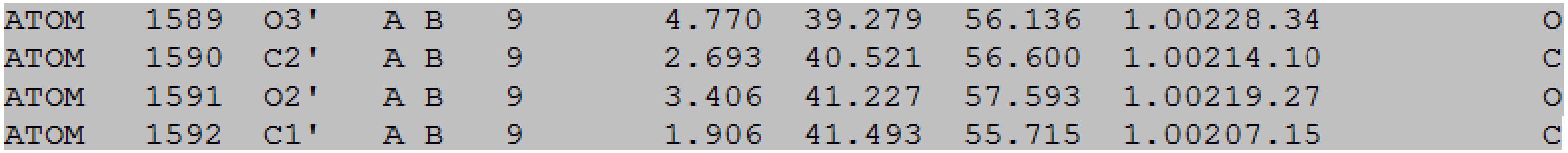
The character ‘^’ is a special character that means to match the beginning of the line

Lesson 4.8: 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
```

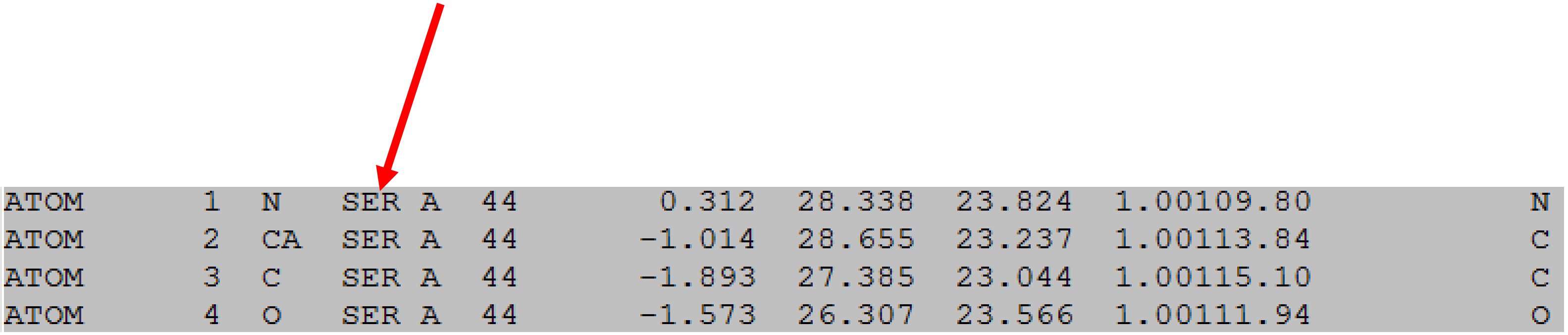


```
ATOM   1589  O3'   A B   9       4.770  39.279  56.136  1.00228.34           O
ATOM   1590  C2'   A B   9       2.693  40.521  56.600  1.00214.10           C
ATOM   1591  O2'   A B   9       3.406  41.227  57.593  1.00219.27           O
ATOM   1592  C1'   A B   9       1.906  41.493  55.715  1.00207.15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

Lesson 4.8: Example

We make a really *complicated* regexp:

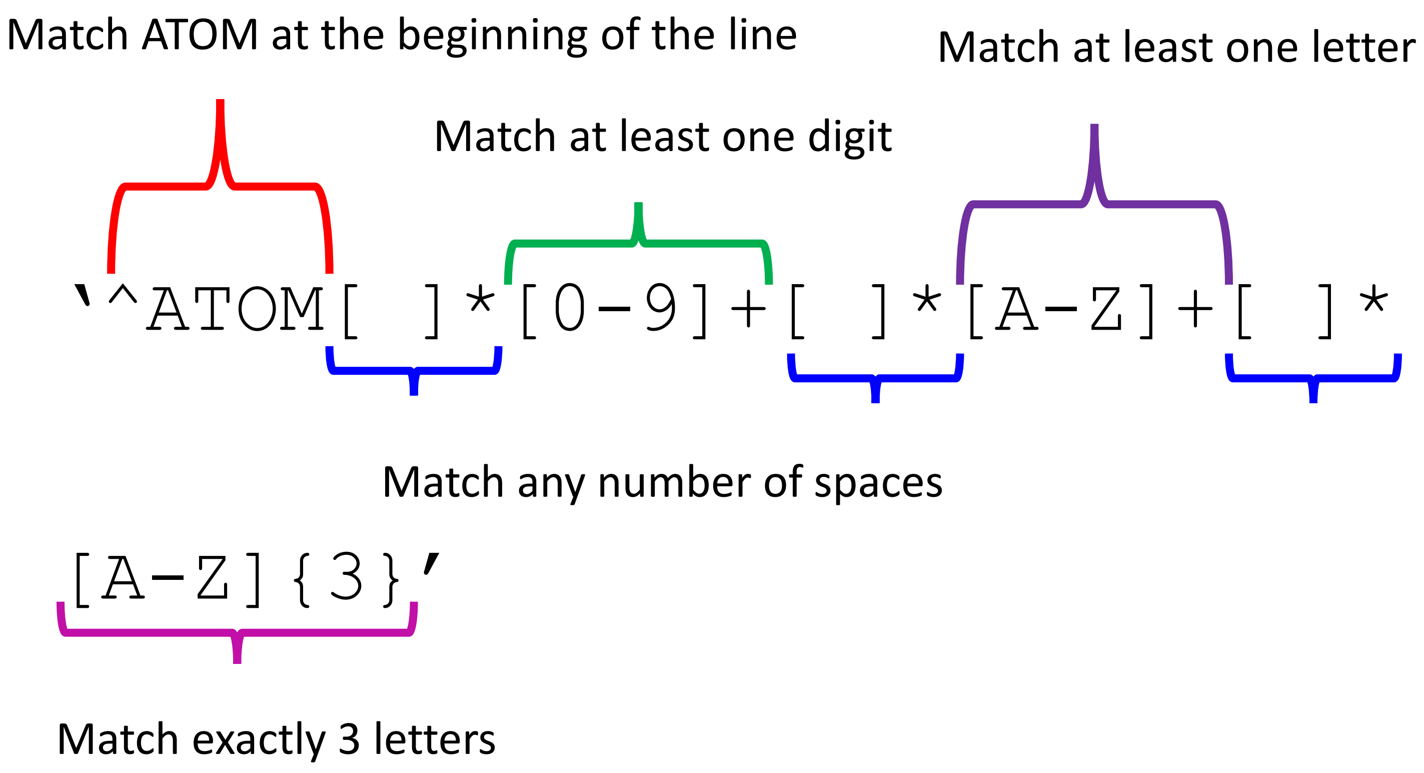
```
egrep ‘^ATOM[ ]*[0-9]+[ ]*[A-Z]+[ ]*[A-Z]{3}’
```



```
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 4.8: 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]+[ ]*
[A-Z]{3}’
```

Match any number of spaces

Match exactly 3 letters

Lesson 4.9:
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 4: Real Life Stuff

Open and edit the file ‘exercise4.sh’

Complete the exercises within.

Then run it.

Lesson 5: Super-basic AWK

Overview:

Lesson 5.1: What is AWK?

Lesson 5.2: AWK Syntax

Lesson 5.1: What is AWK?

AWK is a *programming language* that we can use within bash to add extra power to our scripts

(“AWK” comes from the initials of its creators, Alfred Aho, Peter Weinberger, and Brian Kernighan, not awkward)

AWK is EXTREMELY complicated, so we are going to focus on a few VERY narrow uses, that will hopefully be helpful to some of you

Lesson 5.2: AWK Syntax

Basic syntax:

```
awk ‘condition {procedure}’
```

When condition is TRUE, do procedure

If condition is FALSE, don’t do that procedure

If you want to do a procedure regardless, then leave out condition

```
awk ‘{procedure}’
```

Lesson 5.3: AWK Field Separator

AWK has the ability to automatically break a line into separate columns (or ‘fields’)

By default, fields are separated by whitespaces – tabs, spaces, etc.

The separator or *delimiter* can be changed:

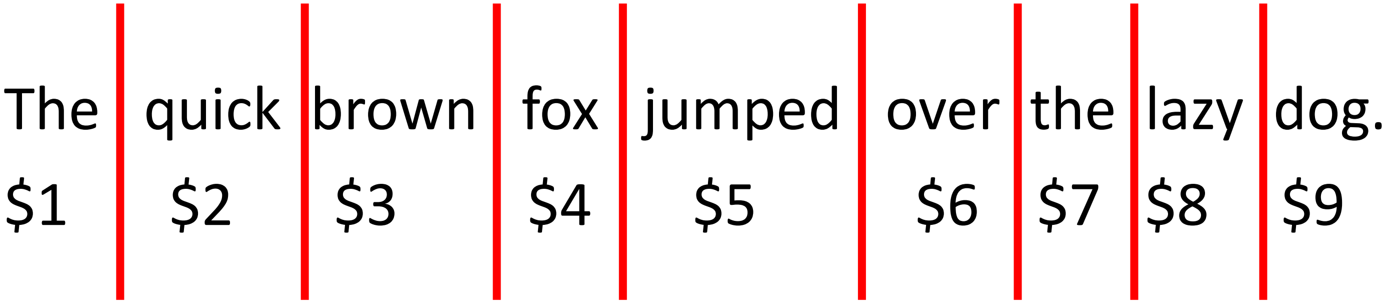
```
awk –F ’delimiter’ ‘condition {procedure}’
<filename>
```

If you want to separate on tabs only:

```
awk –F ‘\t’ ‘condition {procedure}’
<filename>
```

Lesson 5.4: Accessing AWK Fields

When AWK breaks up a line, each field gets its own number:



The	quick	brown	fox	jumped	over	the	lazy	dog.
$1	$2	$3	$4	$5	$6	$7	$8	$9

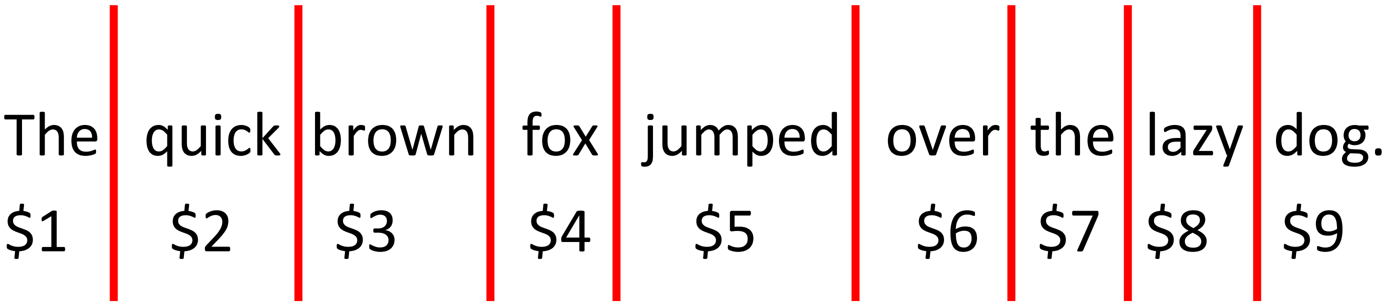
The ‘name’ of each column takes the form of `$number`

Each field can be accessed through its name!

Lesson 5.4: Accessing AWK Fields

Example:

Suppose we have the following sentence in a file called “sentence.txt”



The	quick	brown	fox	jumped	over	the	lazy	dog.
$1	$2	$3	$4	$5	$6	$7	$8	$9

We could access the third word via:

```
awk ‘{print $3}’ sentence.txt
```

Lesson 5.4: Accessing AWK Fields

With this method, we can actually access ‘columns’ in data:

Rank	Title	Studio	Worldwide	Domestic
1	Avatar	Fox	$2,788	$760.5
2	Titanic	Par	$2,186.8	$658.7
3	Star Wars: TFA	BV	$2,059.7	$932.3

We could output the Worldwide gross for each movie using:

```
awk –F ’\t’ ‘{print $4}’
```

Lesson 5.5: AWK Conditions

We can use ‘conditions’ to only access certain columns:

So let’s print the Worldwide gross ONLY for movies made after 2010:

Rank	Title	Studio	Worldwide	Domestic	%
$1	$2	$3	$4	$5	$6

Overseas	%	Year
$7	$8	$9

```
awk –F $’\t’ ???
```

Lesson 5.5: AWK Conditions

We can use ‘conditions’ to only access certain columns:

So let’s print the Worldwide gross ONLY for movies made after 2010:

Rank	Title	Studio	Worldwide	Domestic	%	Overseas	%	Year
$1	$2	$3	$4	$5	$6	$7	$8	$9

```
awk -F $’\t’ ‘$9 > 2010 {print $4}’
```

Exercise 5

Navigate to ~/20170302/exercise5

Move ‘exercise5.sh’ to ~/linux-1/files

Open ‘exercise5.sh’

Complete the instructions, and run the script

Lesson 5.6:
AWK Continuing Education

We can do many, many, many more things with AWK

(Add rows and columns based on conditions, etc.)

It is EXTREMELY powerful, and can get extremely complicated

Hopefully, you now have enough to at least know what questions to ask

Further AWK learning:

https://www.gnu.org/software/gawk/manual/gawk.html

Closing

Thanks for coming!

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