Week 03

ssh, grep, regexes, if, and bash scripts

February 2, 2021

Tonight we'll look at grep a bit more, we'll learn what a regex is, youll see how if statements work in bash, we'll see a tidy way to write bash commands, and you'll use a cool tool called ssh.

---

1 Review of the commands

By this point we've learned a whole bunch of basic command line programs you can use on Linux. Spend a few minutes reviewing them in class.

2 Vim Recap

There is alot to know about Vim, it's a great editor.

The main thing to know is

```
root@machine$ vim filename # this opens a file called "filename"
```

When you are in vim, you need to know two things to get started:

- When you first launch vim you are in NORMAL mode. You cannot type into your file yet.
- Hit "i" on the keyboard to enter INSERT mode. In INSERT mode you can type in your file.
- When you are done working on your file, hit ESC. This puts you back in NORMAL mode
- In NORMAL mode you can save and quit, save, or just quit.
- To save and quit type ":wq".
- To quit without saving type ":q!"
- To save without quitting type ":w". You are still in NORMAL mode. Remember if you want to go back to editing the file, type "i". This will put you back in INSERT mode.

3 Review SSH

We won't go too in depth. If you want to go deep, check out Open SSH Mastery, an awesome book by Michael Lukas.

Here's what I'll tell you though:

- Open a terminal and type "ssh-keygen" to generate a key pair.

- The keypair will be stored in the .ssh folder in your home directory.
- Guard the id_rsa with your life. If anyone gets that file, they can hack your servers.
- If you want to connect to a server with ssh (without using a password), copy the contents of the id_rsa.pub file to a line in the .ssh/authorized_keys file.
- With respect to the last point - I typically put 3 keys on my servers. One for my mac, one for my windows pc, and one for my linux laptop. That way I can connect to my server from any pc in my house. If you use multiple computers, you might want to generate a key pair on each one, and put all the public keys on your server.

Class Example
Have everyone in class send me their pub keys. Put the pub keys on my server. Show they can all connect. This might be chaos with too many students. In the case of many students, just do this with a handful

4 How To Download Files off the Internet

5 grep

5.1 Commonly used grep flags

5.2 Regular expressions and grep

5.3 grep for things that aren't English or Numbers

6 Bash Scripting

7 8:00 - 8:45 Grep

7.1 8:00 - 8:10 grep revisited

Now lets look at some more advanced pattern matching used in Linux via the grep command. Grep stands for Global regular expression print, it uses regular expressions to search for strings. "What's a regular expressions???" - we'll get to those nasty things in a second, but first we'll take a peek at grep.

Now, grep is immensely useful, as we already saw last week. Last week we typed

```
melvyn@thinkpad$ history | grep wget
```

to look through our messy bash history to find exactly the command of interest to us, to see what files we downloaded in the past and to potentially download them again.

Probably 100 times a week I use a command that you're now ready to appreciate:

melvyn@pc$ history | grep ssh

I use ssh all week to connect to a handful of differnt machines and can't remember the ip addressI need to connect to . So I'll just peek in my history and see what machines I connected to in the past and then reconnect using that info.

A poem I like: https://www.cc.gatech.edu/ spencer/poems/woods.txt

You can wget the poem

We are going to cover a bunch of grep options to pick apart this poem.

1. -i *grep -i HARNESS woods.txt*
2. -w *grep -w arness; grep -wi Harness*
3. -v for inverse grep i.e. *grep -v arness*
4. -r mkdir -p a/b; mv woods.txt a/b; grep -r arness *
5. -n grep -rn arness *
6. -o grep -o "
$$a - z$$
+" The o flag means to return only the thing being searched for. Without this flag, grep returns the whole line when it finds a match.
7. grep can match lines after + including pattern 'grep -A1 arness woods.txt '
8. grep can match lines before and including pattern 'grep -B1 arness woods.txt'
9. grep can match lines around pattern 'grep -C3 arness woods.txt'
10. -l to list files containing a pattern *grep -l arness **

remember if we see any irrelevant error messages from grep, we can redirect them to the

```
grep -l arness * 2> /dev/null
```

e.g.

```
$mkdir dir
    $grep -l arness *
    grep: dir: Is a directory
    woods.txt
```

BUT

```
$grep -l arness * 2>/dev/null
lecture.txt
woods.txt
```

A reference for later: https://opensourceforu.com/2012/06/beginners-guide-gnu-grep-basics/

8 8:10 - 8:15 Grep Exercises

Use the above patterns on the file 'hamletSolilquy.txt'. See what patterns you can extract. Make sure you test all of the patterns above, as you need to understand grep very well for your homework!

Convince yourself that the flags I've just shown you work:

- -i
- -w
- -v
- -r
- -n
- -A
- -B
- -C
- -l

then we'll move on to another interesting part of grep.

9 8:15 - 8:35 Grep and regular expressions

Ah, but we have yet to get to regular expressions! Grep stands for **Global Regular Expression Print** or **Generic Regular Expresion Parser** or something or other, but the "RE" definitely stands for Regular Expression.

They say when a programmer has a problem and says "I know, I'll use a regular expression!". This is because regular expressions are tricky and easy to screw up if you don't pay attention. The lesson here is the same as with using vim - don't complain that the thing is hard, just learn to use it and then use it without whining! I suspect this proverb is so popular because alot of people don't pay attention to what a regex (that's short, slang for a regular expression) is and how to use it.

Regular expressions are for pattern matching. They are found in every major programming language out there - C++, Python, Java, etc. and you can use them in the bash shell too along with the 'grep' utility.

`https://www.gnu.org/software/grep/manual/html_node/Basic-vs-Extended.html`

There are two types - regular and extended. We'll just look at the basic ones - extended is about the same. In your free time click the link above and read it quickly. You'll see they are about the same.

Note to students: **as with much of what I'll tell you this semester, I don't have alot of what I'm telling you today memorized. I'm**

vaguely aware of the various symbols I'm going to show you and I often have to look at the documentation before typing anything to amke sure I type it right. It's important that you understand the concept of a regular expression. Then you'll be able to use google to figure out exactly what you need to type.

In basic regular expressions the meta-characters

- '?'
- '+'
- '{'
- '|'
- '(', and
- ')'

lose their special meaning; instead use the backslashed versions

- '\?'
- '\+'
- '\{'
- '\|'
- '\('
- '\)'

NOTE!! I'll repeat for emphasis - we are using basic regular expressions in this lesson. So if we want to use the '+' meta-character we have to instead type '\+'

What to know:

1. The period (.) matches any single character.
2. ? means that the preceding item is optional, and if found, will be matched at the most, once.
3. * means that the preceding item will be matched zero or more times.
4. + means the preceding item will be matched one or more times.

5. n means the preceding item is matched exactly n times, while n, means the item is matched n or more times. n,m means that the preceding item is matched at least n times, but not more than m times. ,m means that the preceding item is matched, at the most, m times.

Some more syntax:

1. ^(Caret) = match expression at the start of a line, as in ^A will match an A at the beginning of a line.
2. $(Dollar sign) = match expression at the end of a line, as in A$.
3. \(Back Slash) = turn off te special meaning of the next character, as in .

   (Brackets) = match any one of the enclosed characters, as in [aeiou]. Use Hyphen "-" for a range, as in [0-9].
4. . (Period) = match a single character of any value, except end of line.
5. * (Asterisk) = match zero or more of the preceding character or expression.
6. {x,y} = match x to y occurrences of the preceding.
7. {x} = match exactly x occurrences of the preceding.
8. {x,} = match x or more occurrences of the preceding.
9. `[\^ ]` = match any one character except those enclosed in [ ], as in `[\^ 0-9]`.

That's it! So, given the file a.txt (see this directory)
We can do the following

```
$ grep "a" a.txt # Find lines with an a
$ grep "a\?" a.txt # find lines with an optional a.
$ grep "a?" a.txt # find lines containing a?
$ grep "a\+" a.txt # find lines with 1 or more "a"s
$ grep "a+" a.txt # find lines containing "a+"
$ grep "a$" a.txt # find lines that end with a
$ grep "[0-9]$" a.txt # find lines that end with a number
$ grep "^[a-zA-Z]$" a.txt # find lines with one letter.
$ grep "a\{2,\}" a.txt # find lines with 2 or more "a"s
```

9.1 8:35 - 8:45 Grep exercises

Change a.txt and change some of the grep patterns and verify that they work as expected on your system. You might have a weird version of grep installed, so let's make sure grep works the same for all of us.

10 8:45 - 9:30 if/test/comparisons

v

10.1 8:45 - 9:05 Introduction to if in bash

Lots of programming languages have 'ifs'. Java has them, python, C, C++, javascript, and beyond. So does bash. It is a standard thing to do in programming to check if a variable equals something.

The bash syntax for if is

```
if CONDITION
then
 command
fi
```

The tricky bit is getting the CONDITION part right, because it is different in different situations, with different data types. I'll show you a little bit about it right now. And, as always, I'll encourage you to go read more if you want the full story. A good link is somewhere down below.

One more useful piece of information is that bash generally interprets values as strings, unless they can be used as numbers, in which case it assumes they are numbers. https://www.tldp.org/LDP/abs/html/untyped.html.

Comparison operators for numbers in bash are:

- -eq
- -ne
- -gt
- -ge
- etc.

Comparison operators for strings are:

- =
- !=
- etc.

for more information see here `https://www.tldp.org/LDP/abs/html/comparison-ops.html`

```
melvyn@thinkpad$ cat myFirstScript.sh
#!/bin/bash


#x1 and x2 are integers, though they could also be interpreted
    as strings.
x1=1
x2=2
```

```
if [ $x1 -lt $x2 ]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
melvyn@thinkpad$ bash myFirstScript.sh
1 < 2
```

Then this script will fail, because you are using an arithmetic comp on strings:

```
melvyn@laptop$ cat myBadScript.sh
x1=1a
x2=2a
if [ $x1 -lt $x2 ]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
melvyn@laptop$ bash myBadScript.sh
# error
```

You can easily fix this with:

Listing 1: option 1

```
#!/bin/bash

x1=1a
x2=2a

if [[ "$x1" < "$x2" ]]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
```

OR

Listing 2: option 2 for if with strings

```
#!/bin/bash

x1=1a
x2=2a

if [ "$x1" \< "$x2" ]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
```

Tip: The old bash advice is to double quote all variables in bash to make sure they are interpreted as a single value.

10.2 9:05 - 9:15 Exercise!

There are many different shell languages. Among them are:

- bash
- dash
- zsh
- csf
- fish

On Ubuntu, when you type **sh** you are actually using **dash**. In the git-bash shell on windows you are using bash. On OS X I think by default you get bash. There are ways to change the shell you use. We aren't concerned with that right now.

Your exercise is to run all the four scripts I just showed you and see fi they work. If they do not, it is likely because you are not using bash. Try running the scripts like : 'bash scriptName.sh' and then 'sh scriptName.sh'

10.3 9:15 - 9:20 Final thoughts about bash

notice the difference between how this script runs with bash and with sh

sh is the bourne shell, bash is the bourne again shell

So I've shown you how to use if and some comparisons and highlighted some pitfalls, okay?

Notice how I made variables and how I used them in here. Now I want to show you a bit more about variables in bash. Again, what I'm showing you will work with sh (probably) but I take no responsibility for it if it doesn't. There are a million shells out there - a cool one I came across recently is fish. I think you install it with apt-get install fish, or fsh, can't remember, but I guess it purports to be a beginner friendly shell. I've never had any issue with bash, I've used the c shell maybe once or twice on an old server, and zsh a few times, but I like bash.

Bash only knows strings. EVERYTHING IN BASH IS A STRING, ALL VARIABLES are STRINGS. Whenever a variable can be treated as an integer, that is bash making a special exception for you. And BASH cannot do decimals. So everything in bash is a string. If it can think about a quantity as an int, it might if you ask nicely . But it will never interpret 1.1 as a number.

10.4 9:20 - 9:30 Another exercise?

Write an if statement to find which is bigger 1.1 or 1.2. You will find you can't do it with the numeric operators, only the string operators.

11 9:30 - 9:35 What did we learn today?

Discussion

12 9:35 - 9:45 Discuss homework

Look through the homework assignment and discuss it.

Remember in the homework that **uniq** only works with sorted input, so when you go to use **uniq** you need to first do something like this:

```
melvyn@thinkpad$ STUFF | sort | uniq
```

13 Fluff

In case extra time, I solved a little problem today using tools you know.

I'm programming a little AVR microcontroller, and the tool you use to program it is avrdude. If you type

```
melvyn@laptop$ avrdude -c ?
```

it will list all the available hardware programmers out there. I wanted ot see if my mkii was supported by this tool so I ran

```
melvyn@laptop$ avrdude -c ? | grep mkii
```

No results. Well I didn't know if avrdude uses upper or lower case, so I just said like this:

```
melvyn@laptop$ avrdude -c ? | grep -i mkii
```

Still no results. Weird. But then I thought, how many types are there?

```
melvyn@laptop$ avrdude -c ? | wc -l
```

No results. But the command definitely outputs results. Do you know how to fix the above commands?

ANSWER: avrdude -c ? writes to stderr, not stdout (for some reason). So I just route my stderr to stdout and then use the piped commands (pipes work on stdout, not stderr:

```
melvyn@laptop$ avrdude -c ? 2>&1 | grep -i mkii
# It comes out!
```