

# LIS-5364

## Crash Course in Shell Scripting; Bash shell

# Multiple commands, one line

& - Run both simultaneously

&& - Run the first one, and then the second  
ONLY IF the first “succeeds,” otherwise stop.

;- - Run the first one, then the second  
regardless of what happens.

# Pipes and redirects

Default behavior:

read from “stdin”, write to “stdout”

- > (over)write/replace a file
- >> write to/append to file
- < read from file
- | pipe output from first command into 2<sup>nd</sup>
- tee pipe AND write to stdout

# BASH

## BASH (Bourne Again) Shell

Lots of “tricks” are available here, eg

- Tab completion
- Up arrow key for history
- Ctrl-R to search history

and many MANY more

# More BASH

*Furthermore, you can modify this environment to fit your needs, via:*

`.bashrc`

(stuff here will be run everytime you open a terminal)

*A great example is the “alias” command. If a command doesn't exist for what you want to do, just ,ake up your own!*

```
alias viewcnn='firefox http://cnn.com'
```

# Extending BASH

Recently, a HUGE uptick in very cool instant shell ideas and replacing some of the older tools, e.g.

fzf

ripgrep(rg)

ALTERNATIVELY – using alternate shells, fish, zsh, etc.

# Linux/Unix Commands

An action or program that a computer can do

Find them with “apropos,” learn about them with “man”

(check these out <http://www.oreillynet.com/linux/cmd/> )

Commands can optionally have ARGUMENTS, in the form of:

## OPTIONS

one dash + letter (ls -a)

two dashes + words (sort --reverse)

## EXPRESSIONS

text; numbers; files; streams – things to be manipulated

# Opening Files

## IN TERMINAL

```
less
```

```
cat (stdout)
```

## COMMAND/ARGUMENT STYLE

```
gnome-open file
```

```
vim textfile
```

```
firefox localfile.html
```

```
firefox http://slashdot.org
```



# SORT

- - i = case INSENSITIVE
- - r = REVERSE
- - g = numbers
- - R = random

# GREP (line matching)

```
grep OPTIONS PATTERN (FILE)
```

Can search over FILES or STDIN

Also, can search ONE FILE or MANY (check -d or -R)

useful flags:

-i (case insensitive)

-v (invert search/show NON-matches)

-l (just show matching FILES, not lines)

# FIND (files)

Searches directory tree rooted at given filename (default current)

Good if you also want to use parameters like “date”, “last accessed”, “size” and so forth.

Often used with `-name` or `-iname`

Also, consider “locate” (database must be setup beforehand)

# SED (stream editor)

Considered an entire language

Usually used with “s” for substitution

Delimiters are usually slashes but can be anything

REGULAR EXPRESSIONS

```
echo "Good day" | sed 's/day/night/'
```

<http://www.grymoire.com/Unix/Sed.html>

<http://sed.sourceforge.net/sed1line.txt>

# AWK

```
awk <search pattern> {<program actions>}
```

Also a text-processor, good for flat-file databases

Also, an entire language

```
awk ' /apples/ { print $2 " " $1 } '
```

<http://www.vectorsite.net/tsawk.html>

<http://www.pement.org/awk/awk1line.txt>

# CLI v GUI?

- Command Line Interface
- 
- Vs
- 
- Graphical User Interface

.....why not both?

# CLI, but GUI-ish

- Nano
- Mc (midnight commander)

Generally, “ncurses” type apps

# From CLI to GUI

- Opening file on command line

```
firefox home.html
```

(Remember, closing the terminal will also close the program, unless you say otherwise, e.g. with “nohup”)



# From GUI to CLI

- Nautilus scripts?

(These are surprisingly HARDER to come by as time goes on...)

# Shell Scripting

So far, we've been doing everything on the command line. What if we want to do 2 or more things?

`&` - run another command **SIMULTANEOUSLY**

`&&` run another command after the first has completed successfully.

`;` - run another command after the first has completed regardless of outcome

# Scripting cont'd

Well, you could also put a bunch of commands in one file, one command per line, and then run that file.

But it's not really “programming (?)”

(yeah, yeah it is)

# Bash scripting

Start with a SHEBANG!

```
#!/bin/bash
```

(note, # is also the comment delimiter)

End by saving AND chmodding

```
chmod +x scriptname.sh
```

Note, they must be run with a FULLPATH!

NOT `scriptname.sh`

But `/home/user/scriptname.sh`

(which can be shortened to `./scriptname.sh` )

# Variables

Set them without \$, use them with \$ (NO SPACES)

```
thingtoecho="Hey, this will be echoed"  
echo $thingtoecho
```

Or "read" them

```
read $yourname  
echo "Whattup $yourname"
```

# Notes on quotes

“Double Quotes” – Print contents, expand variables

'Single Quotes' – Print contents LITERALLY

`backticks` – Execute command, put contents in quotes\*

(also, backslashes “literalize” special chars)

```
date="eh, whenever"
```

```
echo "date"      echo 'date'      echo `date`
```

```
echo "$date"    echo '$date'    echo ` $date `
```

*(my advice: don't actually use backticks. More on that later)*

# Obvious Use of Variables

```
echo "Hey, so, what's your username?"
read username
echo "you know, while you're at it, might as well give me your password."
read password
echo "WOW, so your username is $username and your password is $password."
echo "thanks, sucker!"
# Internal storage
echo "New entry:" >> "/home/class/ListOfSuckers.txt"
echo "Username:$username" >> "/home/class/ListOfSuckers.txt"
echo "Password:$password" >> "/home/class/ListOfSuckers.txt"
# The below will add a newline for us, to keep them visually separated.
echo "" >> "/home/class/ListOfSuckers.txt"
```

# But lets clean up...

```
suckerlist="/home/class/ListOfSuckers.txt"
echo "Hey, so, what's your username?"
read username
echo "you know, while you're at it, might as well give me your password."
read password
echo "WOW, so your username is $username and your password is $password."
echo "thanks, sucker!"
# Internal storage
echo "New entry:" >> $suckerlist
echo "Username:$username" >> $suckerlist
echo "Password:$password" >> $suckerlist
# The below will add a newline for us, to keep them visually separated.
echo "" >> $suckerlist
```



# Special Variables

Environment Variables – usually capitalized, contain system/shell info

SHELL, HOME, PATH (?), LOGNAME etc

Argument Variables

\$1 is first, \$2 is second, and so on.

\$# is number of args, \$\* is all of them

## Create commands with argument variables

Contents of the file, apologize\_to.sh

```
echo "I'm sorry about all the  
password stuff, $1"
```

Usage: apologize\_to.sh USERNAME

# Making decisions (if then)

(you can consolidate with ;)

```
if condition
  then
    Do this
  elif (else if)
    then Do this other thing
  else ( everything else was false)
    Do this other other thing
fi
```

# Expressing conditions

True = 0. False = 1

Generally, just use double brackets.

```
If [[ $variabletotest = "thing you want" ]];
```

```
if [[ $password == "password" ]]; then
```

```
    echo "Wow. You officially have the worst password in the world."
```

```
else
```

```
    echo "Well, at least you're not using the worst password in the world."
```

```
fi
```

# Conditions with commands

Another slick way to use conditions is with commands. The general rule is, if a command **HAS** a result, it's true, if not, it's false.

```
if grep "fish" petlist.txt; then
    echo "Looks like you got a fish!"
else
    echo "sorry, no fish here"
fi
```

# While

“While” is very similar to if; it keeps repeating the loop while a condition is true.

(while + read + “cat pipe” or “<”) is very good for reading files

# For loop

For VARIABLE in (RANGE or LIST)

do

done

# What I didn't (and likely won't) cover

Functions (actually, pretty useful)

Arrays

Bash Pattern Matching

Signal Catching (what to do with kill)

Traps (untimely stopping/catching vars)

Using sed / awk / grep in your scripts