Shell scripting

CIS 2230 Linux System Administration

Lecture 11

Steve Ruegsegger

Review

- Name 3 'popular' linux text (non-gui) editors
- What are the 4 modes of vi?
- How do you move between them?
- How do you exit vi with and without saving a file?
- What does $ emacs -nw do?
- How do you exit emacs?
- What is keystroke for the “undo” command in emacs?
- What is the default GUI editor in Ubuntu?

What Are Shell Scripts?

- In the simplest terms, a shell script is a file containing a series of commands.
- The shell reads this file and carries out the commands *as though* they have been entered directly on the command line.

Review

Script File Format

- Simplest example:

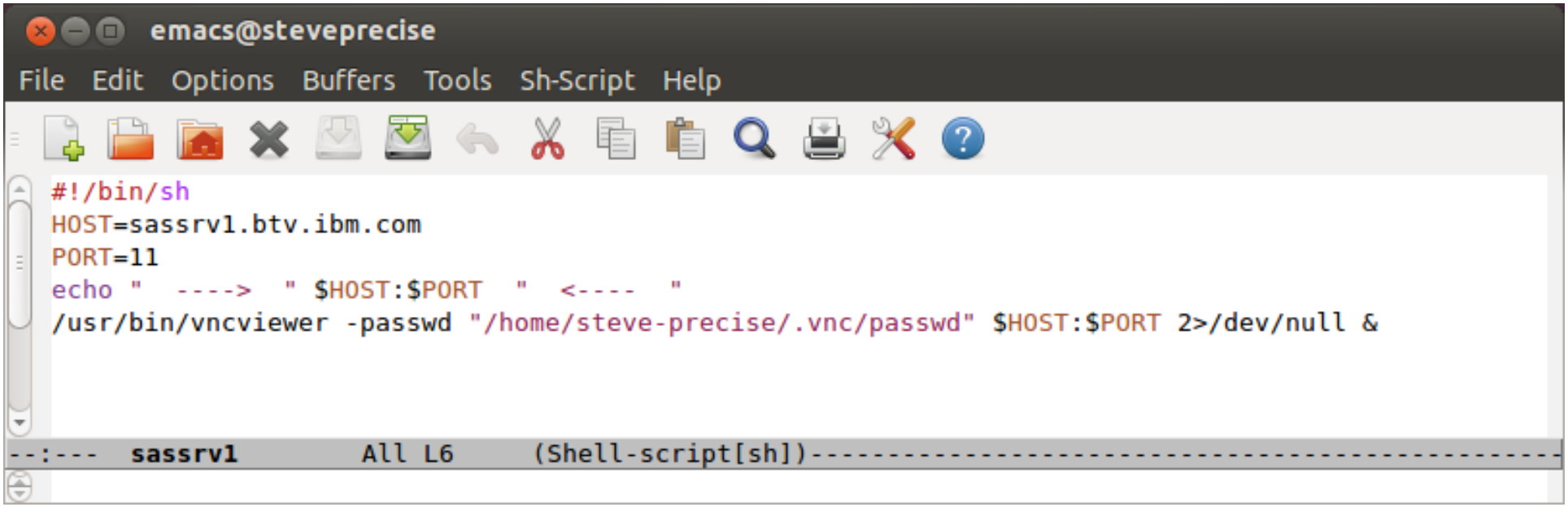
```
#!/bin/bash
# This is super simple
echo 'Hello world'
```

- The first line of our script is a little mysterious.
  - It looks like it should be a comment
  - The `#!` character sequence is, in fact, a special construct called a *shebang*. (Slang for "hash-bang")
  - The shebang is used to tell the system the name of the interpreter (shell) that should be used to execute the script that follows.
  - *Every* shell script should include this as its first line.

Review

How To Write A Shell Script

- Write the script in an editor
  - Shell scripts are ordinary text
- "Execute" it



Review

How to 'execute' the script

- 2 methods

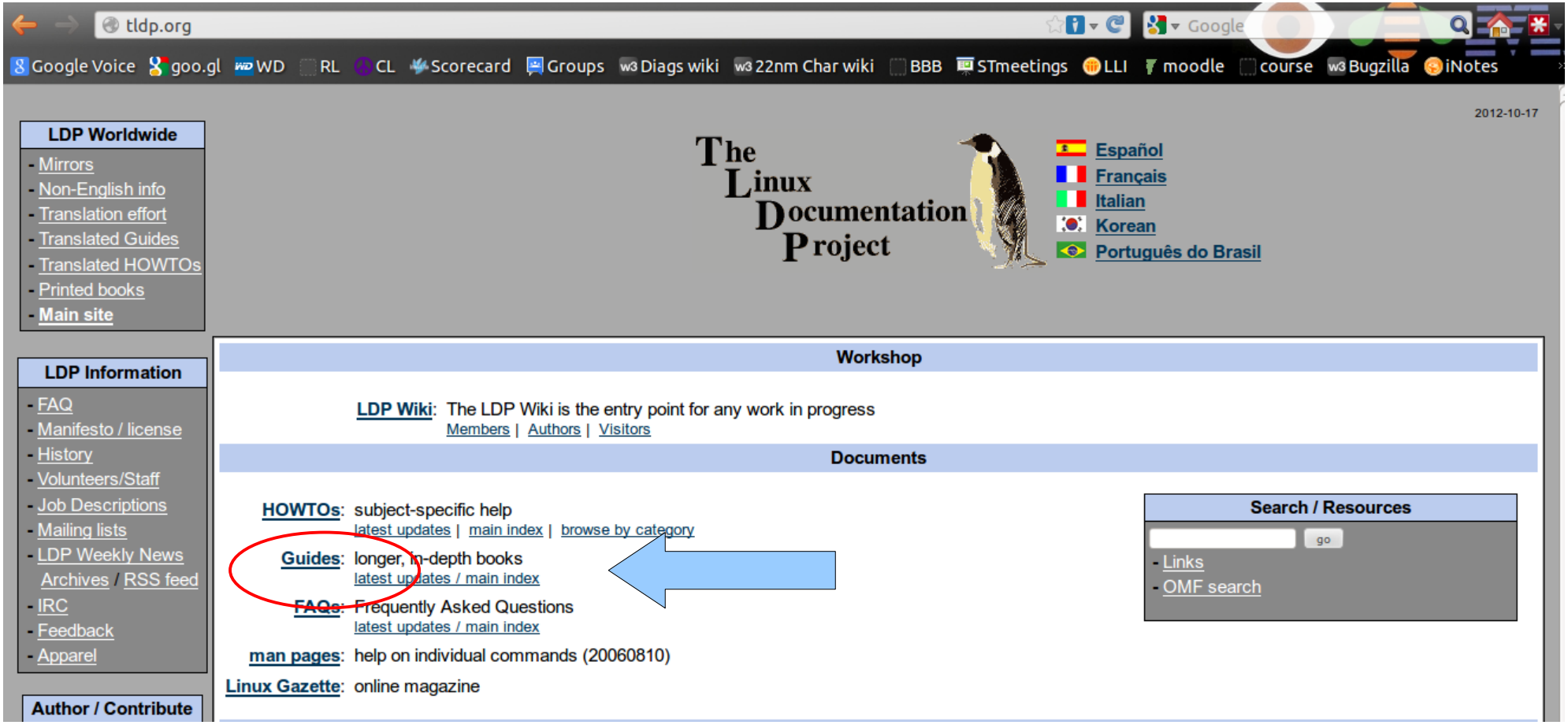
1. Method #1
   - Just put commands in a file
   - The file is argument to a shell
2. Method #2
   - Change file permissions to be executable by user
     - (We'll cover this in detail later)

     ```
     $ chmod u+x <file>
     ```
   - Run your script file from the command prompt
     - Understand: Explicit vs Implicit :)

Review

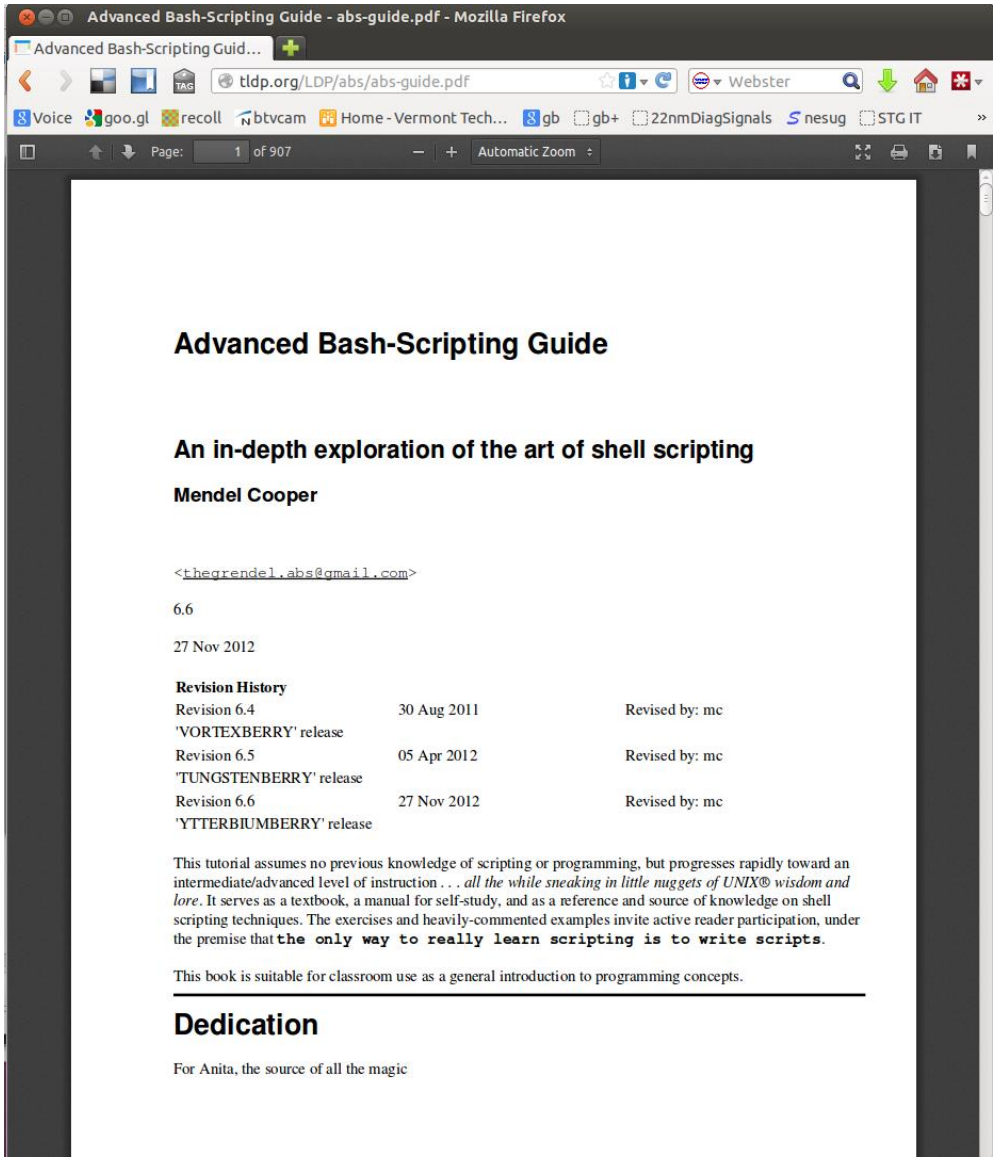
Getting help on bash programming

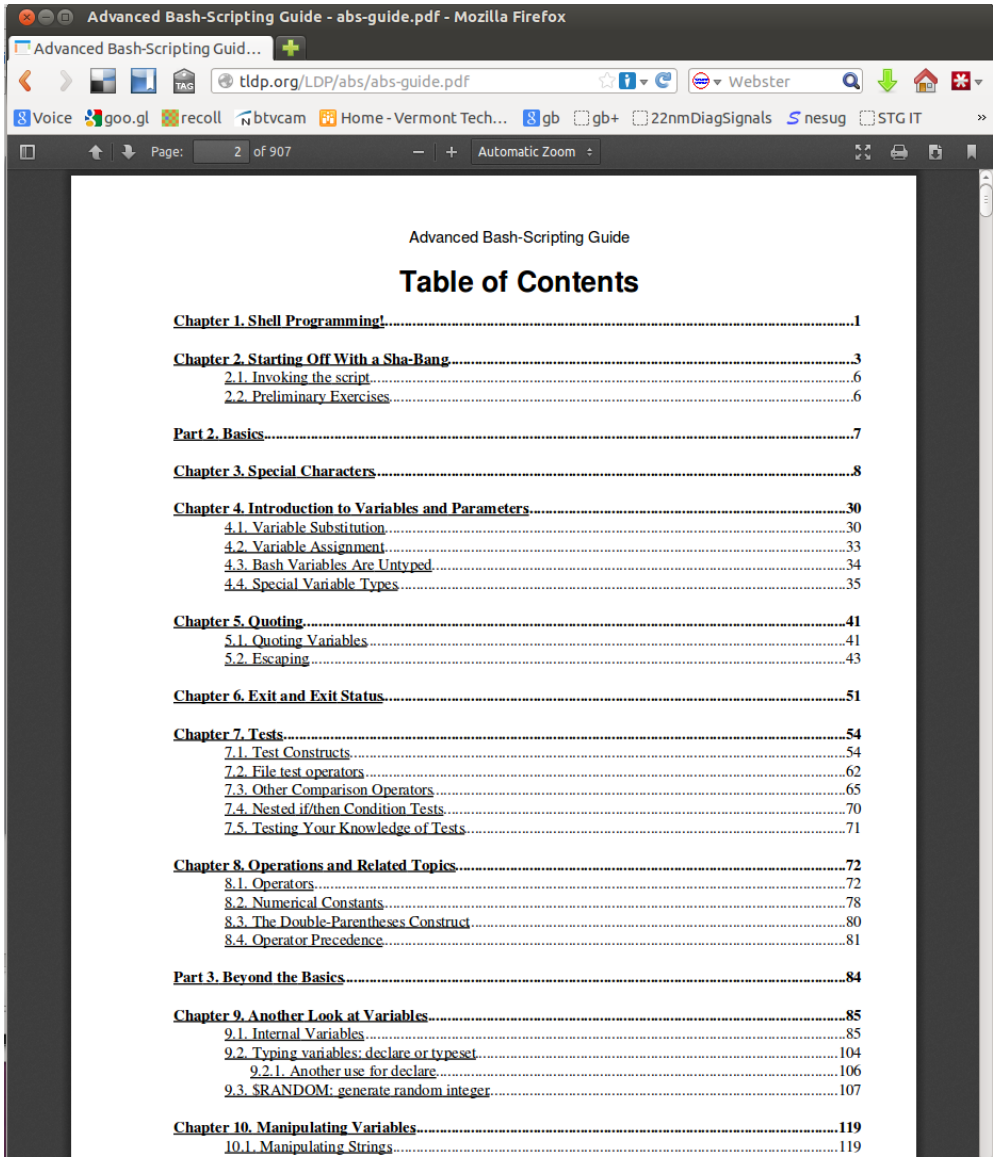
- tldp.org – The Linux Documentation Project



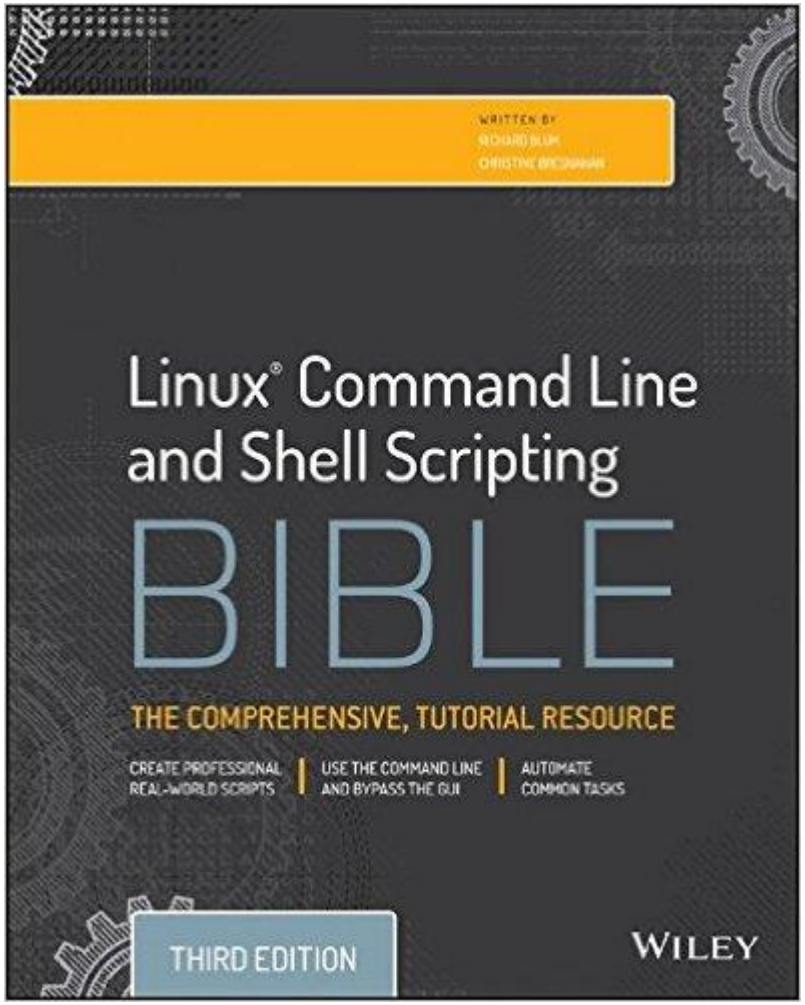
tldp.org/LDP/abs/html

ABS = Advanced Bash-scripting guide





Another good resource



Customer Reviews

13

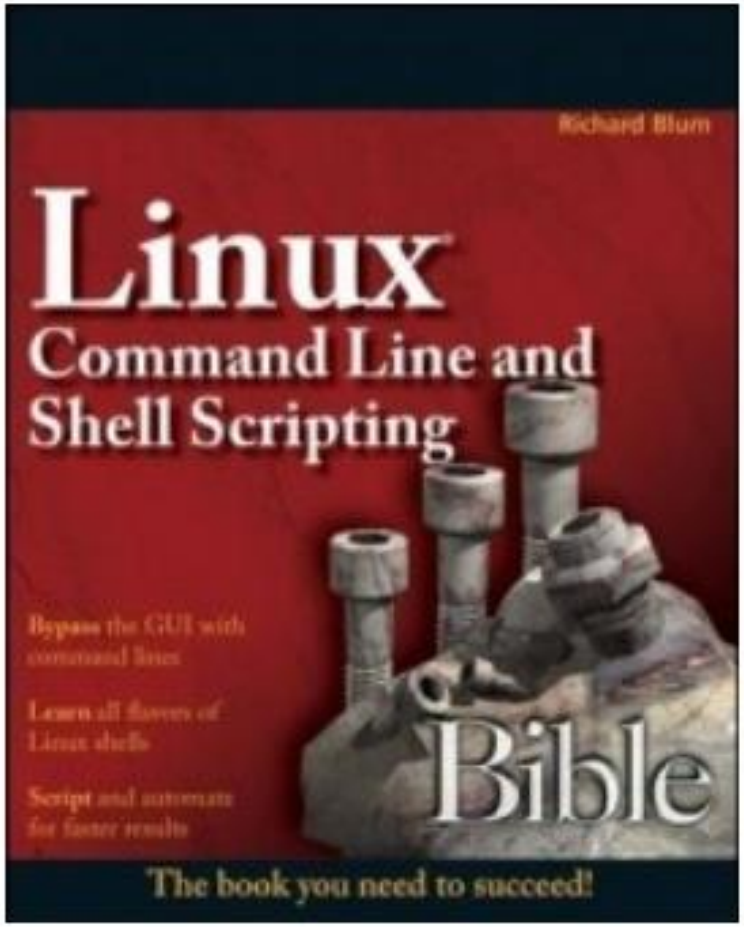
4.8 out of 5 stars

5 star	85%
4 star	15%
3 star	0%
2 star	0%
1 star	0%

See all 13 customer reviews

The first edition is a 'free' PDF

Linux Command Line and Shell Scripting Bible



Author:	Richard Blum
Isbn:	978-0-470-25128-7
Year:	2008
Pages:	809
Language:	English
File size:	19.9 MB
File format:	PDF
Category:	Linux & Unix

Contents of first edition

Lecture 5

Lecture 6
Lecture 13
Lecture 10 – vi, nano,emacs

Lecture 11

Lecture 9

7 Key programming concepts

1. Assigning variables (ch 8)
   - 1) string, 2) command substitution, 3) arithmetic
2. If-then-else
3. test expressions

(ch 9)

4. While looping
5. For looping

(ch 10)

6. Passing arguments
7. Reading keyboard input

(ch 11)

Chapter numbers from 'free PDF" book on page 10
"Linux Command Line and Scripting Bible" 1st ed.

1. Assigning Variables – strings

- Simply:
  - variable=value
  - No spaces around the =. (try it)
  - The shell does not care about the type of data assigned to a variable. It treats them all as *strings*
- Examples

```
a=5
a=foo.bar
b="a string"        # Embedded spaces must be within quotes.
c="a string and $b"     # Other expansions such as
      # variables can be expanded into the assignment.
f="\t\ta string\n"      # Escape sequences such as tabs and
      # newlines.
```

Assigning Variables – strings

- Use quotes for strings

```
$ z=this is a test
is: command not found

$ z="this is a test"
$ echo $z
this is a test
```

Assigning Variables – command substitution

- Capture command substitution
  - $( ) or backticks
  - We've used this inside commands before
  - But it works the same to assign a var value

```
d=$(date "+%F")              # d = 2012-10-17
cp fstab fstab.$d
```

Assigning Variables – Integer Math

- 4 ways to do *integer* math:
  1. `expr` – the original and the trickiest. Use inside command substitution. No quotes. *Must* escape symbols to keep from the shell
  2. `$[ ]` – short cut and no escaping symbols
  3. `let` – this command includes the assignment
  4. `$(( ))` - integer arithmetic expansion (like let)
- Examples: (*How many minutes in 4 hours?*)

```
$ hrs=4
$ min=`expr $hrs \* 60`
$ min=$[ $hrs * 60 ]
$ let "min = $hrs * 60"
$ min=$(( $hrs * 60 ))
```

Assigning Variables – Floating pt Math

- Float Math!
- `bc` – *built-in* bash calculator command
  - Build the expression in a *string*
  - echo the string and pipe to bc
  - Scale = 0 by default for division (not sure why)
- Example (convert feet to meters)

```
$ ft=5.8
$ m=$(echo "scale=2; $ft * 12 * 2.54 / 100" | bc)
```

Assigning Variables - arrays

- Arrays
  - Initalize with parens OR brackets
  - Dereference (“use”) with `${}`

```
MYARRAY=( 4027 1749 1713 3050 )
r=$RANDOM
let "n = ($r % 4)"
s=${MYARRAY[$n]}
echo r=$r n=$n myarray=$s

→ r=20399 n=3 myarray=3050
```

array

Same as:

```
n=$(( $r % 4 ))
n=`expr $r % 4`
```

Variable names

- Where does a variable name *end*?
- *e.g.* I want to rename a file to one ending with a "1"

```
$ fn="foobar"
$ mv $fn $fn1
mv: missing destination file operand after `foobar'
Try `mv --help' for more information.

$ echo mv $fn $fn1
mv foobar
```

- Solution → put curly brackets {} around var name

```
$ mv $filename ${filename}1
```

Here Documents or heredoc

- When there is a large text:
- A here document (a.k.a. heredoc) is a form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command.
- It works like this:

```
command << token
text
token
```

- "token" is any unique string you want, which simply does not exist in the "text"
- I like EOP, which I always consider End-Of-Print.

Examples

```
#!/bin/bash
# Script to retrieve a file via FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
bye
_EOF_
ls -l $REMOTE_FILE
```

```
#!/bin/bash
# Script to write a new script

cat >> "new_script.sh" <<EOP
#/bin/sh
find . -mtime +30 -delete
EOP
```

Heredoc's are *not really* used for env. vars

```
$ text=$(cat <<EOT
> here is some text
> lots of lines
> now I'm done
> EOT
> )

$ echo "$text"
here is some text
lots of lines
now I'm done
```

Shell Functions

- There are two different formats for defining functions:

```
function name {
commands
return
}
```

- and

```
name () {
commands
return
}
```

- Called simply by the function name

2. Flow Control: branching with if

- You all understand:

```
if <expression> then <command>
else <command>
```

- The shell script format is:

```
if evalcmd; then
commands
[elif expr; then
commands...]
[else
commands]
fi
```

syntax

- Memorize that the semi (;) acts the same as a new line.
- Both newline and ; separate commands
- Therefore, the 3 codes below are the same
- I will use both ; and newline interchangeably.

```
if <eval> ; then
   command1
   command2
fi
```

```
if <eval>
then command1
     command2
fi
```

```
$ if <eval> ; then command1; command2; fi
```

Linux shell exit status

- The evalcmd in an if-then is the exit status from *any* linux command
- An exit status is define to be 0 for "success" and *any other number* for an error.
  - Notice, that exit status is 0 for good, anything else is bad.
  - Most T/F flags are 0 for F and anything else for T
- Therefore if/then construct tests whether the exit status of a list of commands is 0, and if so, executes one or more commands
- The exit status is stored in the $? env variable from "all" commands

Example "if" statements right from prompt

- Try these academic example :

```
if pwd ; then echo yes; else echo no; fi
if xxx ; then echo yes; else echo no; fi
```

Any "command" to be evaluated
The "if" checks the return code for any ERROR. "0" is means "no error"

- Or this:

```
#/bin/bash

username=jimbob

if grep $username /etc/passwd
then
        echo "username $username found"
        echo "the .b* files are"
        ls -a /home/$username/.b*
else
        echo "the username $username was not found"
fi
```

3. Most used evaluation command → test

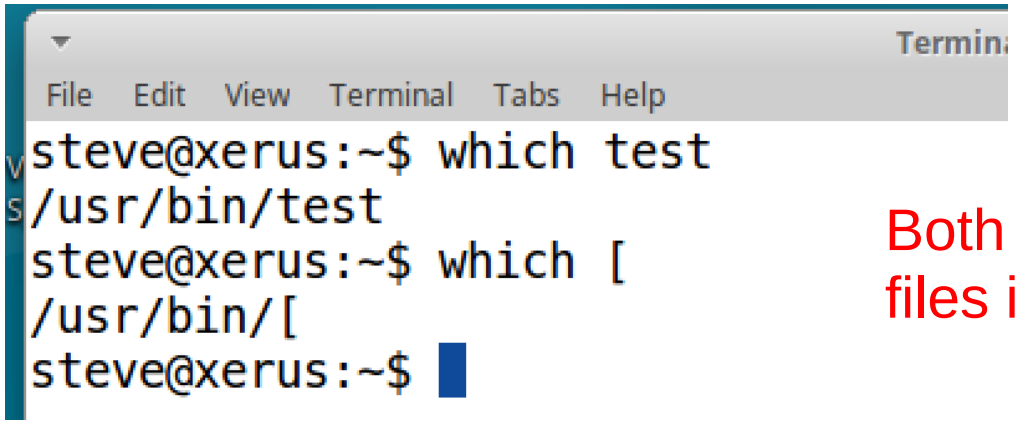
- By far, the command used most frequently with "if" is `test`.
- The `test` command performs a variety of checks and comparisons.
- It has two equivalent forms:

```
test expression
```

- and the more popular format:

```
[ expression ]
```

Same!



Both test and [are executable files in /usr/bin

The test command evaluates an 'exit' error

- Remember! The `test` command returns an “exit status”
  - an exit status of zero (0) when the expression is true
  - an exit status of anything other than zero when the expression is false.
  - Huh?
    - I think it’s kinda strange
    - It's opposite of T/F
    - Think of it as an “error” code. 0 = no error (= true)

3 types of `test` expressions

- There are 3 classes of test conditions:

1. Numeric
2. String
3. File

- Each class above has it's own set of commands!

1. Numeric expressions for test

TABLE 9-1

The test Numeric Comparisons

Comparison	Description
n1 -eq *n2*	Check if *n1* is equal to *n2*.
n1 -ge *n2*	Check if *n1* is greater than or equal to *n2*.
n1 -gt *n2*	Check if *n1* is greater than *n2*.
n1 -le *n2*	Check if *n1* is less than or equal to *n2*.
n1 -lt *n2*	Check if *n1* is less than *n2*.
n1 -ne *n2*	Check if *n1* is not equal to *n2*.

"numeric" test get the **2 letter** version

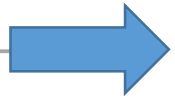
```
n=`who | wc -l`
if [ $n -gt 3 ] ; then
   echo "who else is on?"
   who
fi
```

2. String expressions for test command

TABLE 9-2

The test Command String Comparisons

Comparison	Description
str1 = *str2*	Check if *str1* is the same as string *str2*.
str1 != *str2*	Check if *str1* is not the same as *str2*.
str1 < *str2*	Check if *str1* is less than *str2*.
str1 > *str2*	Check if *str1* is greater than *str2*.
-n *str1*	Check if *str1* has a length greater than zero.
-z *str1*	Check if *str1* has a length of zero.



"string" test get the **symbol** version

```
top=$(top -d4 -n1  | head -8 | tail -1 | awk '{print $13}')

if [ $top = 'firefox' ];  then
   echo "Lots of firefox here"
elif [ $top = 'compiz' ]; then
   echo "the GUI is the biggest hog!"
fi
```

Warnings on strings in [ ]

String order

Trying to determine if one string is less than or greater than another is where things start getting tricky. There are two problems that often plague shell programmers when trying to use the greater-than or less-than features of the `test` command:

- The greater-than and less-than symbols must be escaped, or the shell will use them as redirection symbols, with the string values as filenames.
- The greater-than and less-than order is not the same as that used with the `sort` command.

Linux Command Line and Shell Scripting Bible v2 (pg 309)

```
$ cat test9b
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing

if [ $val1 \> $val2 ]
then
   echo "$val1 is greater than $val2"
else
   echo "$val1 is less than $val2"
fi
```

escape, else > is redirect to a file.

More warnings on strings in [ ]

- Don't get burned by using string compare operators (>,=,<) for *numbers*! (I have...)
- *Understand* this output:

```
steve@xerus:~$ a=5
steve@xerus:~$ b=11
steve@xerus:~$ if [ $a \> $b ]; then echo "$a is > $b, really"; fi
5 is > 11, really
steve@xerus:~$
```

3. File expression for test command

TABLE 9-3

The test Command File Comparisons

Comparison	Description
`-d file`	Check if *file* exists and is a directory.
`-e file`	Checks if *file* exists.
`-f file`	Checks if *file* exists and is a file.
`-r file`	Checks if *file* exists and is readable.
`-s file`	Checks if *file* exists and is not empty.
`-w file`	Checks if *file* exists and is writable.
`-x file`	Checks if *file* exists and is executable.
`-O file`	Checks if *file* exists and is owned by the current user.
`-G file`	Checks if *file* exists and the default group is the same as the current user.
file1 `-nt` *file2*	Checks if *file1* is newer than *file2*.
file1 `-ot` *file2*	Checks if *file1* is older than *file2*.

memorize ↑

"file" test get the **dash-letter** version

example

```
#!/bin/bash

# test-file: Evaluate the status of a file

FILE=~/.bashrc

if [ -e "$FILE" ]; then
      if [ -f "$FILE" ]; then
            echo "$FILE is a regular file."
      fi
      if [ -d "$FILE" ]; then
            echo "$FILE is a directory."
      fi
      if [ -r "$FILE" ]; then
            echo "$FILE is readable."
      fi
      if [ -w "$FILE" ]; then
            echo "$FILE is writable."
      fi
      if [ -x "$FILE" ]; then
            echo "$FILE is executable/searchable."
      fi
else
      echo "$FILE does not exist"
      exit 1
fi

exit
```

An *improved* version of the test command: [[ ]]

- Of course, someone *improved* on `test`
- Recent versions of bash include a compound command that acts as an enhanced replacement for `test`.
- It is a bash symbol and not an executable command
- It uses the following syntax:

  `[[ expression ]]` ← Notice the "spaces!"

- where, like `test`, expression is the exit status of a linux command.
- The `[[ ]]` command is very similar to `test` (it supports all of its expressions)
- Benefits:
  - You don't have to *escape* the string compare: > or <.
  - It adds 2 cool new operators →

Regex in test [[ ]] using =~

1. regex pattern matching for strings!

```
string1 =~ regex
```

- which returns 0 (no error) if string1 is matched by the extended regular expression regex.
- Extended regex pattern definition
- No quotes around the regex (seems odd to me)
- Example:

```
A="steve steven mary Stephen Joe Steve"
for name in $A; do
  if [[ $name =~ [Ss]te(v|ph)en? ]] ; then
     echo "name=$name"
  fi
done
```

Globbing in test [[ ]] using ==

2. File globbing!

- Another added feature of [[ ]] is that the == operator supports filename matching the same way as shell 'globbing'
- This makes [[ ]] very useful for evaluating file and path names
- No quotes around the glob (still seems odd to me)
- *Don't confuse with regex, they are similar, but globbing is different*
- Example

```
if [[ $file == l*.txt ]] ; then
     mv $file /nfs/backup
fi
```

(( )) - Improved integer arithmetics

- bash also provides improved integer math with the (( )) command
- It supports a full set of *arithmetic* evaluations:
  - >, <, == symbols rather than -gt, -lt, -eq from ( )
  - Pre and post-increment and -decrement:
    `var++, var--, --var, ++var`
  - Logical AND an OR: &&, || rather than -a and -o from ( )
- Examples

```
port=5901
if (( $port > 5900 )) ; then
     vncserver :$port
fi
```

Combining Expressions

- It's also possible to combine expressions to create more complex evaluations.

Operation	test	[[ ]] and (( ))
AND	-a	&&
OR	-o	||
NOT	!	!

How to test if an integer is within a range.

```
if [ $INT -ge $MIN_VAL -a $INT -le $MAX_VAL ]; then
      echo "$INT is within $MIN_VAL to $MAX_VAL."
else
      echo "$INT is out of range."
fi
```

Summary of Brackets

- `$( )` - command substitution
- `$[ ]` - same as `expr`, original integer math
- `$(( ))` - integer math expansion, like `let`. An improvement on `$[ ]` and `expr`
- `if [ ]` - same as `test` command
- `if ()` - original integer math
- `if [[ ]]` - improved [ ] or test command
- `if (( ))` - improved ( ) integer math

A nice *summary* of brackets and test [ ] expressions!
http://tldp.org/LDP/abs/html/refcards.html

4. Looping

- While looping:

```
while <condition is true>; do
    commands
done
```

- Same <condition> as if-then.
- The $? exit status is tested
- You can use test or [[ ]] in the while condition

```
#!/bin/bash

# while-count: display a series of numbers

count=1

while [ $count -le 5 ]; do
      echo $count
      count=$((count + 1))
done
echo "Finished."
```

Original test [ ]

Improved Int. Math $(( ))

Looping

- Until looping:

```
until <expression is false>; do
      commands
done
```

- Example

```
#!/bin/bash
# until-count: display a series of numbers
 count=1
 until [[ $count > 5 ]]; do
    echo $count
    let "count=$count+1"
 done
 echo "Finished."
```

Improved test `[[ ]]`

Improved Int. Math `let`

Be sure to compare this script to the previous one

Breaking the loop

- `break` command
  - You can break-out of a loop at any iteration
  - Stops the loop right then. Continues after the “done”
  - Works with while and until
  - If nested, then it “breaks out of” the inner most loop

```
for (( a=1; a<10; a++ )) ; do
   echo "** $a **"
   if [ $a -eq 6 ]; then
       break
   fi
done
```

Skipping in the loop

- `continue` command
  - Just skips to the next “iterator”
  - Doesn't finish that one loop, and jumps to the next loop
  - “stays in” the loop, just skips one “beat”

```
for (( a=1; a<10; a++ )) ; do
   if [ $a -eq 6 ]; then
       continue
   fi
   echo "** $a **"
done
```

5. Flow Control: Looping With for

- Syntax:

```
for <var> in <list>; do
    commands
done
```

- Typical sources of <list> :

```
for s in fr so jr sr; do
for n in `seq 1 20`; do
for i in {A..D}; do
for file in ~/labs/lab*.txt; do
```

Manual list
Numeric list
Char list
File glob

  - Note: in order for "for" to use the file *glob*, it must see a glob wildcard: * or ?.

More for syntax examples

A list of files

```
for f in file1 file2 file3 file5
do
 echo "Processing $f"
 # do something on $f
done
```

You can also use shell variables:

Shell variable list

```
FILES="file1
/path/to/file2
/etc/resolv.conf"
for f in $FILES
do
        echo "Processing $f"
done
```

You can loop through all files such as *.c, enter:

Glob!

```
$ for f in *.c; do echo "Processing $f file.."; done
```

More on that `for` list:

- Reading from a command

```
for product in $(cat /usr/inventory.txt);
do
    echo "product $product"
done
```

- The command might get complicated! You can separate out if you want to.

```
dirs=$( cat /etc/passwd | awk -F: '($3 >= 1000){print $6} )
for dir in $dirs; do
    echo "** $dir **"
    ls $dir
done
```

A new favorite technique of mine

More on that for <list>: IFS

- What if the list is not space-separated?
- You can change what does separate a list.
- It's the `IFS` env var - Internal Field Separator !
- The default is `$' \t\n'` which means space, tab and newline.
  - Note: $'somestring' is a 'string' of literals
- *e.g.* if you are looping on names which could contain spaces, then remove the space from IFS

```
IFS=$'\n\t'
for student in $(cat /usr/students.txt); do
   echo "** $student **"
done
```

IFS

- Don’t break on spaces; break on colon!

```
IFS=:
A="New York:New Hampshire:Vermont:South Carolina"
for state in $A; do
    echo state=$state
done
```

```
steve@xerus:~$
steve@xerus:~$ IFS=:
steve@xerus:~$ A="New York:New Hampshire:Vermont:South Carolina"
steve@xerus:~$ for state in $A; do
>     echo state=$state
> done
state=New York
state=New Hampshire
state=Vermont
state=South Carolina
steve@xerus:~$
```

Run without IFS to see what happens

Reset IFS by: `$ IFS=$' \n\t'`

C-type for loops allowed in bash

- Another format for for
- Looks like a loop from C or perl

```
for (( a=1; a < 10; a ++ ))
do
   echo ** a = $a **
done
```

6. Passing arguments

- So far, all our variables have been 'hard-coded' in the script.
- A program can get 'live' data in several ways:
  - Read from the keyboard “interactively”
  - Pass to the program when it is executed as “arguments”

Positional Variables or Shell Arguments

- The arguments are passed to a program are *positional variables*.
  - The first argument passed to the script is assigned to the variable $1
  - the second argument is $2,
  - and so on up to $9.
- Notice that the names of the variables are actually the digits 1 through 9
- The positional variable $0 always contains the *filename* of the script.
- Ex: $ myscript.sh steve /home/steve 30

$0 $1 $2 $3

example

```
#!/bin/bash

# traveltime - a program to calculate how long it will
# take to travel a fixed distance

# syntax: traveltime miles mph
# $ traveltime 90 40
# The trip will take 2 hours and 15 minutes

TOTMINUTES=$(( ($1 / $2) * 60  ))  #  miles / mph * 60
HOURS=$(( $TOTMINUTES / 60 ))
MINUTES=$(( $TOTMINUTES % 60 ))

echo "The trip of $1 miles at $2 mph will take $HOURS hours and
$MINUTES minutes"
```

```
$ ./traveltime 90 40
The trip of 90 miles at 40 mph will take 2 hours and 15 minutes
```

Additional positional variables

- `$#` is the number of variables 
- `$*` and `$@` are a list of all positional variables
- Ex: ./script "word" "words with spaces"
- "`$*`" produces a one string result of all args:
  e.g. "word words with spaces"
- "`$@`" produces a two string result:
  e.g. "word" "words with spaces"

Parameter	Description
$*	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character).
$@	Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word surrounded by double quotes.

More than $9?

- The first 9 are easy: $1 to $9
  then use curly brackets: ${10}, ${11}
- Args are space-delimited. Use quotes to group together
- $ checkname steve ruegsegger new hampshire

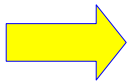
  $0 $1 $2 $3 $4

- $ checkname "steve ruegsegger" "new hampshire"

  $0 $1 $2

- The `shift` command moves each arg “down one”.
  - It's like shift @ARGV; in C or perl
  - After a shift, $2 becomes $1, $3 becomes $2, etc.

7. Reading Keyboard Input



```
#!/bin/bash
# read-integer: evaluate the value of an integer.

 echo -n "Please enter an integer -> "
 read int
 if [[ "$int" =~ ^-?[0-9]+$ ]]; then
   echo "Input value is indeed an integer." >&2
 else
   echo "Input value is not an integer." >&2
   exit 1
fi
```

Example – this amount of formatting should be memorized

```
#!/bin/bash

if [ $# -eq 0 ]; then
   echo "./afs [start|restart|stop]"
   exit 1;
fi


if [ "$1" == "start" ]; then
    echo "starting openafs-client service"
    sudo /etc/init.d/openafs-client force-start
    echo "klog ruegsegs (using getpw)"
    klog ruegsegs -pipe `~/getpw -id ruegsegs -key afs get`
fi

if [ "$1" == "stop" ]; then
    echo "stopping openafs-client service"
    sudo service openafs-client stop
fi

if [ "$1" == "restart" ]; then
    echo "RE-starting openafs-client service"
    sudo /etc/init.d/openafs-client restart
fi

exit
```

Skill: Editing a string variable

- We use piped commands to 'edit' a string variable
  - use `echo` to 'print' out the variable to stdout
  - use `sed` to edit the variable (stdin to stdout)
  - Put that in *command substitution* to set to a new, editted variable
- e.g.

```
oldname=year2014
newname=$( echo $oldname | sed 's/2014/2015' )
cp /home/steve/$oldname /home/steve/$newname
```

- What if you had a *comma* separated file and wanted to loop on the fields? The loop below doesn't work. One way to fix is with IFS. However, you can also fix it using sed.

```
line=a,b,c,d
for element in $line; do
    echo one element is: $element
done
```