Bash Scripting

MAKING THE ADMIN'S LIFE
THAT MUCH EASIER

About the Instructor

- - nathan.isburgh@edgecloud.com
- Unix user 15+ years, teaching it 10+ years
- Unix Administration and Software Development
 Consultant

- Goofy, forgetful (remember that)

About the Course



Hours: 8:30 - 5:00

■ Lunch: | 1:45 - 1:00

Breaks about every hour

Throw something soft at me if I get too long in the tooth

□ Telephone policy

• Take it outside, please

Restrooms

Across from central stairs

■ Refreshments

 Downstairs in break room, mini-fridge in classroom, machines by stairs

About the Students

- Name?
- □ Time served, I mean employed, at Rackspace?
- Department?
- General Unix skill level? What about Linux?
- ⊕ And familiarity with Bash?
- How do you use Linux in your position?
- What are you hoping to take away from this class?

Expectations of Students

- Strong foundation in basic Linux use and administration
 - Preferably through RHCSA
- Strong understanding of working in the shell
- Ask Questions!
- Complete the labs
- Email if you're going to be late/miss class
- Learn something

Intentionally Left Blank

Scripting Basic Concepts 2 - Scripting Basic Concepts

Overview

- There are several basic concepts about the shell and scripting which must be understood before tackling more complex problems
 - Basic shell syntax
 - Shebang syntax
 - Quoting
 - Exit status and subprocesses
 - Variables
 - Commenting

Shell Syntax

- Shell scripting is simply placing a sequence of shell commands into a file, for future "playback"
 - Obviously there are plenty of details, which is what we will be exploring in this course
 - At the end, though, it all boils down to shell commands
- Therefore, it follows that you must already have a strong foundation in basic shell syntax
 - Quoting
 - Environment variables
 - Commands

Scripting 101

- Simple shell scripts simply run command after command,
 as if the user typed them in at the command line
 - More complex shell scripts actually make decisions about what commands need to be run, and might even repeat certain sequences to accomplish a given task
- Scripts start executing at the top and stop when there are no more commands to execute or when exit is called
 - Or due to a syntax error!

Example



```
echo "Hello, what is your name?"
read NAME
echo "Hello $NAME, it's nice to meet you!"
echo -n "The current time is: "
date
```

- Using the echo command, this script asks a question.
- The read command accepts input from the user and stores it in the environment variable NAME
- The script finishes up with a couple more echo statements, greeting the user and announcing today's date

Running The Example

- If we put the example in a file called myscript, we can execute the script as:
 - ⊕ bash myscript
- Which instructs your interactive shell to start a new shell, bash, to open myscript and execute each line as if the user had typed it in manually
- - Reading each line of the file, bash would interpret the words and perform the given action
- There are many interpreted languages available for scripting, including all of the shells, python, ruby, perl, etc.

Interpreters

- Following this idea, to run a script, you simply feed the file to the appropriate interpreter
 - ⊕ bash mybashscript
 - perl myperlscript
- This works fine, but sometimes it's more user-friendly to allow the script to be run directly, removing the need for an external call to the interpreter...
 - ⊕ ./mybashscript
 - myperlscript
- How is this done?

Shebang!

- This is accomplished with the shebang (#!), also known as a hash bang, pound bang or hashpling.
- The basic idea is very simple
- When the kernel is asked to execute a file, the content must either be machine code (compiled software), or a file that starts with the shebang sequence
- If the <u>first two characters of the file are a hash mark and an exclamation mark</u> (shebang!), the rest of the line is expected to be a pathname for an interpreter, which will then be invoked to "run" the file as a script
 - Connecting the script to stdin of the interpreter process

Back to the Example

So, add an appropriate shebang to the example:

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
echo "Hello $NAME, it's nice to meet you!"
echo -n "The current time is: "
date
```

Then add execute perms so the script can be run directly:

```
[root@localhost ~]# chmod a+x myscript
[root@localhost ~]# ./myscript
Hello, what is your name?
Linus
Hello Linus, it's nice to meet you!
The current time is: Sun Jul 21 09:39:33 CDT 2013
[root@localhost ~]#
```

Details to Note

- Note the use of quoting in the example
 - Remember that everything in a shell script must follow shell syntax!
- If something would need to be quoted on the command line (due to whitespace or metacharacters), it will also need to be quoted in the shell script
- In addition to single and double quotes, remember your escape character: \ (the backslash)
 - Do you know the difference between the quoting mechanisms?

Exit Status

- Another important detail to internalize when shell scripting is the importance of exit codes (or statuses)
- Every single time a process is finished executing, it notifies the kernel via an exit system call
- There is a required parameter to the exit system call, known as the exit status
- The exit status is a number, and there are only two values meaningful to the kernel and shells:

 - Non-Zero: Any non zero exit status implies a failure of some sort

Exit Status and Scripting

- The reason that the exit status is so important to shell scripting is because all of the shell features used in scripting are based on exit status
 - Conditionals
 - Looping
 - Intelligent command separators
- Note that the actual non-zero values a program might use, such as 14, -8, 2, etc, do not have standard meanings
 - The documentation for an application might specify the meaning of particular exit codes, which can then be checked in a script through the \$? special environment variable

Variables

- Variables in shell scripting are nothing more than standard environment variables
- This is convenient; the known rules and capabilities apply
 - NAME=value
 - NAME="quoted value"
 - ls \$NAME
 - echo Hello \${NAME:-Sir/Madam}, may I help you\?
- The set and env commands are useful
- See bash manpage under heading "Parameter Expansion"

Commenting

- Commenting falls under the larger topic of coding style,
 which could be a class unto itself
 - Note that style is an individual attribute, developed over time as a software developer
 - It is also often lightly or strictly specified by organization
- To simplify this discussion, let us recall the Golden Rules of Commenting...

The Golden Rules of Commenting

- Always comment code which is not obvious to a nonauthor reader
 - You should not comment "i=i+1"
 - You should comment "rsync -vazpc \$WHAT \$WHERE"
- Always comment functions: their purpose, use, arguments, expectations and results
- Always comment the overall program's purpose and behavior at the top of the file
 - Include dates and authors (maybe an abbreviated revision history?)
- Always comment when not sure if you should
 - They don't cost anything!

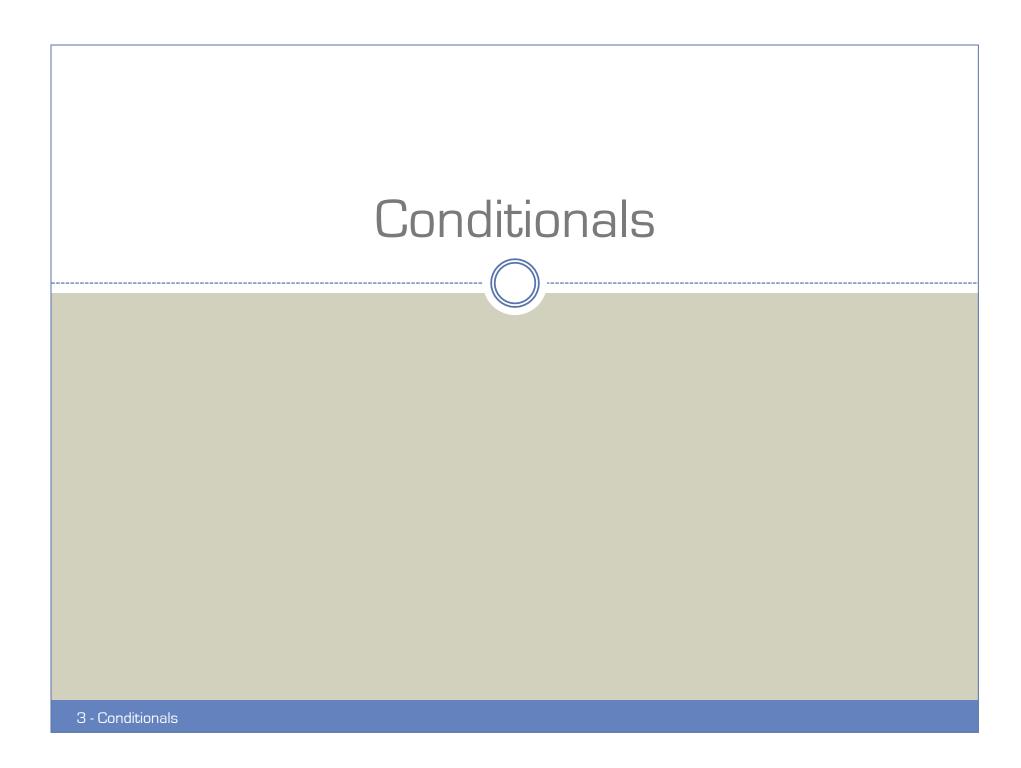
Lab

Write a basic "Hello world" shell script

- The script should greet the user by name, then welcome him to the world of scripting. Consider commands or environment variables which might obtain the user's login name.
- Match the following output format, substituting the underlined values appropriately:
 - ⊕ Hello <u>nisburgh</u>. Welcome to the world of scripting.
 - ⊕ The current date is Monday, July 22, 2013.

Follow all of the guidelines discussed

- Make it a standalone executable using the shebang syntax
- Comment appropriately
- Read documentation for assistance



To Execute or Not To Execute

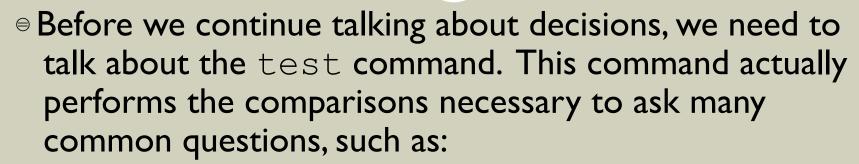
More advanced problems require the script to make decisions. There are two basic ways to make decisions with shell scripts:

- The most basic and powerful conditional
- "If some condition is true, then do these things"

case statements

- A streamlined version of an if statement, mainly used to improve readability and maintenance of code
- "Taking a given input and several possible values I'm interested in, which one matches? Then do these things based on that match"

The test Command



- The result of the test is in the exit status
 - True Exit 0
- See the man page on test for additional details and more flags; there are many tests it can perform

The if Statement

```
if list
    then list
    [ elif list
        then list ]
    ...
    [ else list ]
fi
```

Example

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
if [ "$NAME" = "Linus" ]
then
   echo "Greetings, Creator!"
elif [ "$NAME" = "Bill" ]
then
   echo "Take your M$ elsewhere!"
   exit
else
   echo "Hello $NAME, it's nice to meet you!"
fi
echo -n "The current time is: "
date
```

This script bases it's response on the name given

The case Statement

```
case word in
  pattern ) list;;
...
```

28

esac

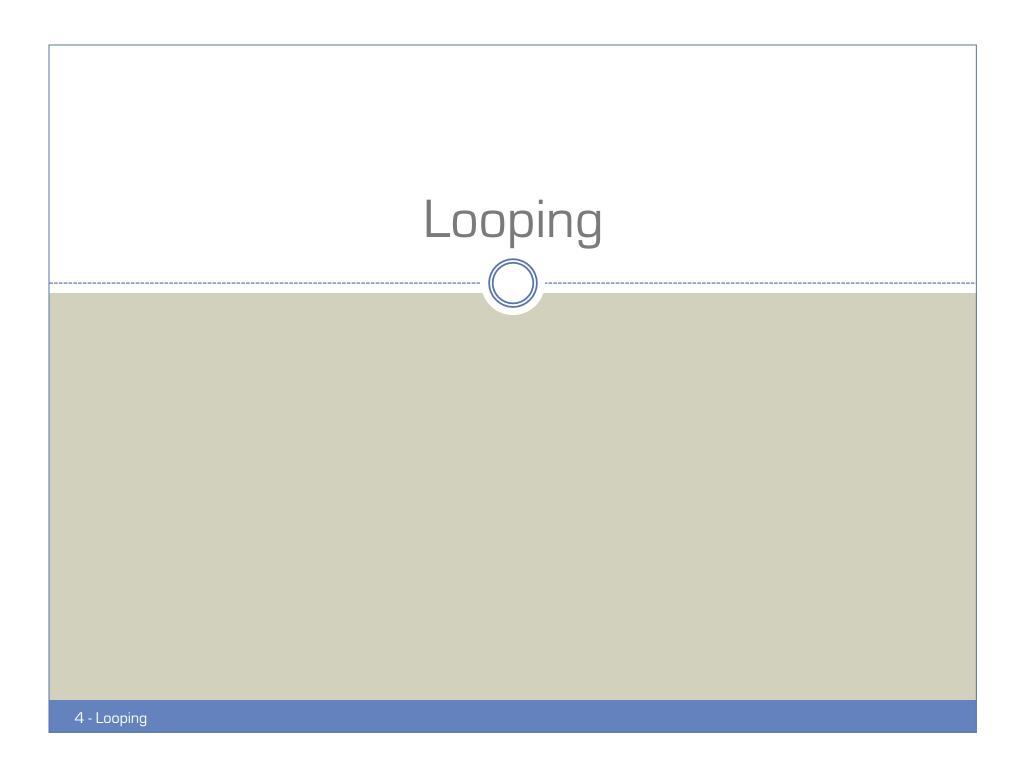
Example

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
case $NAME in
  "Linus" )
    echo "Greetings, Creator!"
    ;;
  "Bill" )
    echo "Take your M$ elsewhere!"
    exit
    echo "Hello $NAME, it's nice to meet you!"
esac
echo -n "The current time is: "
date
```

This script maintains identical behavior, but uses a case statement

Lab

- Write a shell script which uses an if statement to print a special message on the first and fifteenth of the month:
 - If it is the first or fifteenth of the month, the script should print:
 - ⊕ YAY! Payday!
 - Otherwise, it should print:
 - ⊕ Boo.. Not yet payday..
- - Check the first, second, tenth, eleventh, fifteenth, and twenty first
- Remember to comment appropriately



Looping

Sometimes a certain sequence of commands need to be run repeatedly, either for a set number of times or while some condition is true. This is accomplished with:

⊕ while loops

- Most common and powerful loop form
- "Check some condition and if true, run these commands. Then check again and if still true, run these commands again. Repeat until the condition is no longer true."

for loops

- Simple method for looping a given number of times or over a list
- "Do this X times."
- "Do this for each item in a list"

The while Loop

- The while loop is the most common, but be aware it has a brother, the until loop
 - The until loop is identical in operation, but the conditional requirements are reversed; execute while the conditional is false
- Basic while/until syntax:

```
while list;
   do list;
```

done

Example

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
while [ "$NAME" != "Linus" ]
do
    echo "I don't know that person, what is your name?"
    read NAME
done
echo "Greetings, Creator!"
echo -n "The current time is: "
date
```

This script will loop until the given name is "Linus"

The for Loop

- There are two major forms of the for loop
- Basic syntax of the first:

```
for (( expr1 ; expr2 ; expr3 ))
    do list;
```

done

Example

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
for (( I=0 ; I<3 ; I++ ))
do
    echo "Hello $NAME!!"
done
echo -n "The current time is: "
date</pre>
```

 This goofy script repeats your name 3 times before giving you the date and time

The for Loop

- Basic syntax:

```
for name in word ...;
    do list;
done
```

Example

```
#!/bin/bash
echo "Hello, what is your name?"
read NAME
for TIME in Three Two One
do
   echo "$TIME"
   sleep 1
done
echo "Hello $NAME!!"
echo -n "The current time is: "
date
```

- This goofy script counts down "3...2...I…" then yells the given name, followed by the date and time
- Note that you can execute a subcommand with the back quotes, and each line will become a list item:

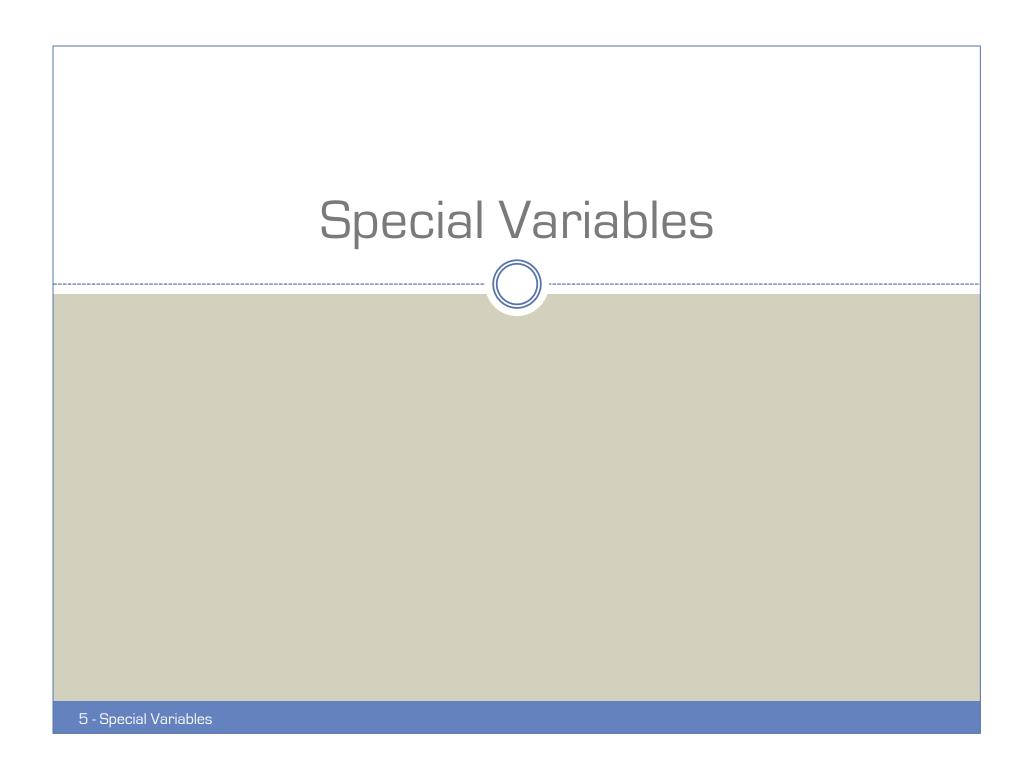
```
for item in `ls /tmp`
```

Lab

- Write a script which uses loops and conditionals to announce every minute as it strikes
- Think of efficient ways to perform this operation, such as sleep statements. Do not "spin." Spinning is when a program runs as fast as it can in a loop waiting on some event to occur, rather than using more intelligent behavior such as alarms, blocks and timers to conserve CPU resources

Intentionally Left Blank

40



Special Variables

- The shell has many special variables to contain information
 - Positional parameters (arguments)
 - Exit status of previous command
 - Bash information
- There are also several ways of getting at the values of variables, known as parameter expansion

Positional Parameters



- The positional parameters are the arguments to the script or a function
- They are assigned numerically, left to right
 - ⊕ script argA argB argC
 - \$0 is the script name
 - \$1 is argA
 - \$2 is argB
 - \$3 is argC
- Also, there are a couple of related special variables
 - \$# is the total number of arguments (not including \$0)
 - \$@ expands to a space separated list of all arguments

Exit Status



- The exit status of the previously executed command can be obtained through the \$? variable
- It is important to consider the meaning of this variable
- Every time you execute a command, it changes
 - If you echo \$?, by the following line it's different already (the exit code of echo)
- For this reason, you will often see scripters storing the value in another variable for future examination:
 - e command with important exit status
 - ESTAT=\$?
 - e if [\$ESTAT -eq 5] ...

Bash Information

- There are dozens of informational variables which are maintained by bash, including some more useful ones:
 - **HOSTNAME**

 - **UID**
 - **BASHPID**
 - **BASH VERSION**
- For a complete list of variables, see the manpage under various headings, including "Special Parameters" and "Shell Variables"

Expanding Variables

- Variables have several methods of expansion to values
 - \$NAME
 - \$ { NAME } to be more precise, or embed in another term
 - \$ {NAME:-word} will expand to word if NAME is not set or null
 - \$ {NAME:=word} will expand to and assign NAME to word if NAME is not already set or null
 - \$ { NAME : ?word} will fail with an error message of word if NAME is not set or null
 - \$ {NAME:offset:length} fetches length characters from NAME starting at offset
 - ⇒ \$ { #NAME } returns character length for value of NAME
- See manpage under "Parameter Expansion" for complete details and additional options

46

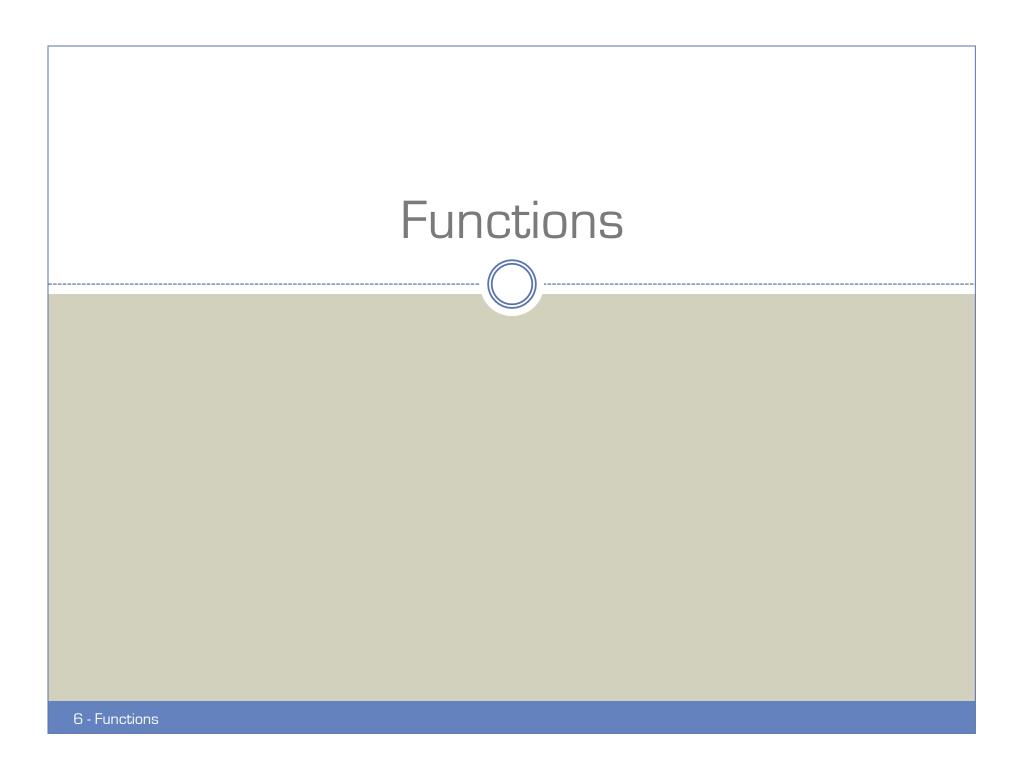
Lab

- Modify the lab from the Loops module to accept two optional parameters
 - The number of total announcements to make before exiting (originally it would run forever, which should be the default)
 - A yes or a no, which indicates whether or not to also print the date with the announcement. Default of yes
- - ⊕ myscript 5 yes
 - Would report 5 times and exit, and each report line would say something along the lines of:

● It is 4:32pm, July 9, 2013!

Intentionally Left Blank

5 - Special Variables 48



Overview

- Functions are an important component of code organization and reuse
- A function allows you to group a series of statements under a name, then *call* the function at any time to execute the collected statements
- You can also pass arguments to the function for it to operate on
- Further, the function can return a value to the caller, indicating status or results

Example

```
#!/bin/bash
sayhello() {
  echo Hello $1
  return 5
}
sayhello Bob
```

- This script defines a function called sayhello, which it then uses to say hello to Bob
- Note how arguments are passed (through standard positional parameters)
- Note how a return value is generated
 - Default is the exit status of last command executed by function

Using Functions

- Functions are often collected in a file, and used by multiple scripts as a library
- To use a library like this, you need to source the file source path-to-library
 - . path-to-library
- For an example, see the startup scripts in the init.d
 folder
- They all use the /etc/init.d/functions library for common operations like starting a service

Lab

- Modify the lab from the Special Variables module such that the reporting functionality is wrapped in one or more functions
- Place the function(s) in a library
- Get creative and add a few more functions to encompass some silly behaviors like using names, printing banners or doing file operations with redirection
- Write a new script which uses the library to offer behaviors to the user through a simple menu system