aws re/start

Bash Shell Scripts

Linux Fundamentals

Name of presenter

Date

What you will learn

At the core of the lesson

You will learn how to:

- Describe common tasks that are accomplished through shell scripts
- Describe basic commands that are frequently included in shell scripts
- Describe basic logical control statements that are frequently included in shell scripts
- Create and run a shell script

What are scripts?

- Text files of commands and related data
- When the text file is processed, the commands are run
- Can be set as scheduled tasks by using **cron**
- Run more quickly by being automated than if they are run manually
- Run more consistently by being automated than manually typed commands
- Easier to ensure that the script has no errors, so the commands run correctly and consistently

Common script tasks

Backup jobs	Log file archiving	System and service configurations	Simplifying repetitive tasks	Anything that can be automated should be

A simple backup script

```
[root@server00 scripts]# cat backup.sh
#!/bin/bash
#Script to back up home directories.

tar -czf backup-job.tar /home/jsouza

echo "Back Up Job Complete at `date`."
```

Script results

```
[root@server00 scripts]# ./backup.sh
tar: Removing leading `/' from member names
Back Up Job Complete at Wed Mar 13 14:21:41 GMT 2019.
[root@server00 scripts]# ls
backup-job.tar  backup.sh  hello.sh  script-template.txt
```

Shell scripts

Process of creating a script:

1. Create script by using a text editor
2. Set script permissions to run
3. Use **./** to run the script

```
#!/bin/bash
echo "Hello!"
echo "Today's date is: `date`"
```

Write the script in the text editor

```
[root@server00 ~]# chmod 744 hello.sh
```

Permissions that are specified for the script file

```
[root@server00 ~]# ./hello.sh
Hello!
Today's date is: Wed Mar 13 13:56:14 GMT 2019
```

Running the script

#!/bin/bash and **#comments**

#! is referred to as *shebang*
First line defines the interpreter to use (give the path and name of the interpreter)
Scripts must begin with the directive for which shell will run them
Location and shell can be different
Each shell has its own syntax, which tells the system what syntax to expect
Example: **#!/bin/bash**
Use **#** to define comments, including the purpose of the script, author information, special directives for the script, examples, and others

First line of the script →

```
#!/bin/bash
echo "Hello!"
echo "Today's date is: `date`"
```

Script template

Some administrators create a script template

- Contains all the relevant information and sections

The template might include:

- Title
- Purpose
- Author's name and contact information
- Special instructions or examples

```
1 #!/bin/bash
2
3 #---------------------------
4 # Author: Chris Mason
5 # Date: 3/12/2019
6
7 #Description: The purpose of this script is to
8 #
9 #
10
11 #---------------------------
12 # Version: 1.2.3
13 #
14 # Special Instructions:
15
16 # Declared Variables:
17
18 #---------------------------
19 # Script Body
```

Script permissions (review)

Script files must have run permissions

chmod 744 hello.sh would permit the user to run the script, but not a group or other users (using absolute mode)

chmod u+x hello.sh would permit the user to run the script, but not a group or other users (using symbolic mode)

```
[root@server00 ~]# chmod 744 hello.sh

[root@server00 scripts]# chmod u+x hello.sh

[root@server00 scripts]# ls -l
total 8
-rwxr--r--. 1 root root  57 Mar 13 13:55 hello.sh
```

Run a script and **./** (review)

- Bash checks for executables along **$PATH**, which does not (and should not) include home directories
- To run a script that is not stored on **$PATH**, use ./ before the script
- Example: **./hello.sh**
  - This example assumes that the script is in the current directory
  - Otherwise, you must provide the absolute or relative path, or maybe an alias (for example, **$HOME**)
  - **cron** jobs always need the full path

```
[root@server00 ~]# ./hello.sh
Hello!
Today's date is: Wed Mar 13 13:56:14 GMT 2019
```

Command substitution (review)

Commands can be placed in the syntax of other commands

Commands are surrounded by backticks (`)

Can be useful in scripts

```
1 #!/bin/bash
2 echo "Hello, $USER!"
3 echo "Today's date is: `date`"
```

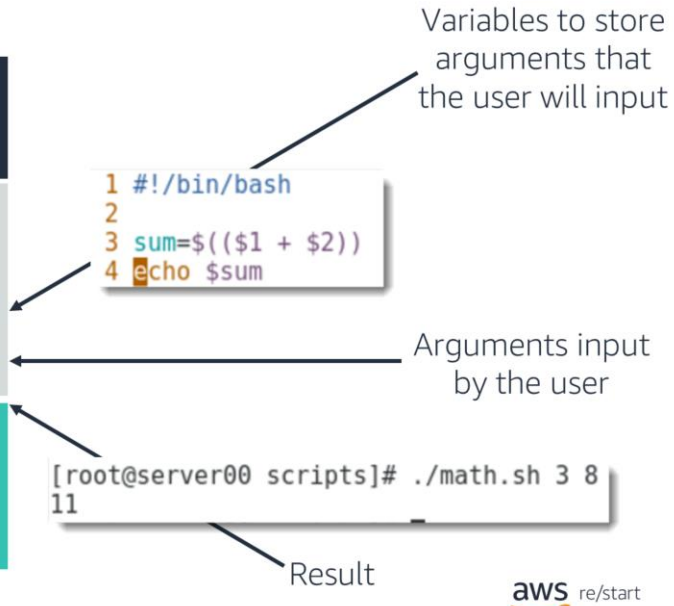
Command substitution

Arguments

Arguments are values that you want script to act on

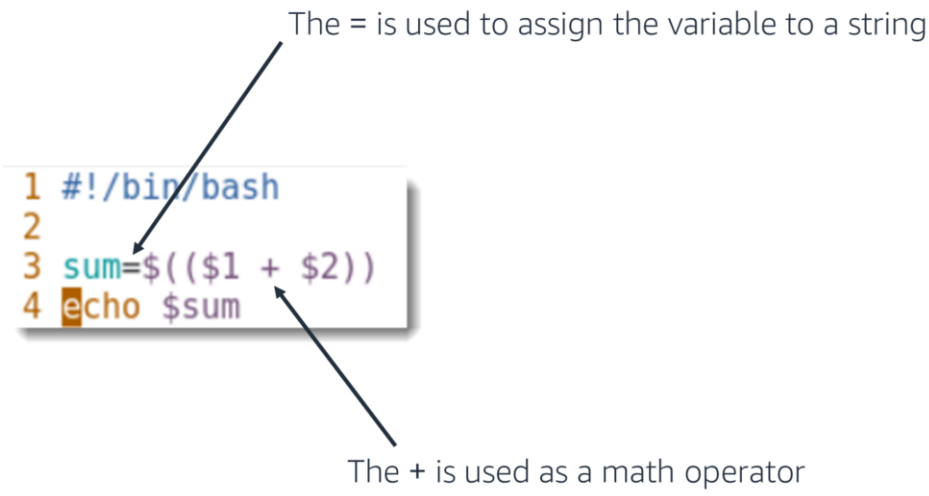
Arguments are passed to the script by including them in a script invocation command, separated by spaces

Example: **$1** is the first argument, and **$2** is the second argument



aws re/start

Operators


The = is used to assign the variable to a string
1 #!/bin/bash
2
3 sum=$(($1 + $2))
4 echo $sum
The + is used as a math operator

Expressions

Expressions are ways to answer questions about what occurs when a script or program runs.

aws re/start

Expressions are ways to answer questions about what occurs when a script or program runs. Such questions might include “Who is running the script?” or “What time is it?”

The answer is usually assigned to a variable for easy reference later in the script. This example asks: “What is the sum of the two numbers that the user entered when they ran the script?”

The value is then stored in the **sum** variable.

The **exit** command

Causes script to stop running and exit to the shell if the previous portion fails

Useful in testing

An unrecognized command

A recognized command after the unrecognized command, but it will not run because of the **exit**

Observe that the script exits at the error

```
1 #!/bin/bash
2
3 sum=$(($1 + $2))
4 echo $sum
5
6 #bad command
7 data
8 exit
9
10 #good command
11 date
```

```
[root@server00 scripts]# ./math.sh 3 8
11
./math.sh: line 7: data: command not found
```

Conditional statements

Different courses of action for a script to take, depending on success or failure of a test

The **rm** command is run only if the **cp** command successfully completes.

```
1 #!/bin/bash
2 # Copy file1 to /tmp
3 # Delete file1 if the copy was successful
4
5 if cp -f file1 /tmp
6 then
7 rm -f file1
8 fi
```

file1 is does not exist.

```
[root@server00 scripts]# ls
backup-job.tar  delete.sh  hello.sh  script-template.txt
backup.sh       file1      math.sh
[root@server00 scripts]# ./delete.sh
[root@server00 scripts]# ls
backup-job.tar  backup.sh  delete.sh  hello.sh  math.sh
[root@server00 scripts]# ls /tmp
file1
```

The **if** statement

If the first command succeeds with an exit code of **0** (success), then the subsequent command runs.

Simplest conditional

```
1 #!/bin/bash
2 # Copy file1 to /tmp
3 # Delete file1 if the copy was successful
4
5 if cp -f file1 /tmp
6 then
7 rm -f file1
8 fi
```

The **if - else** statement

Defines two courses of action:

- If the condition is true (**then**)
- If the condition is false (**else**)

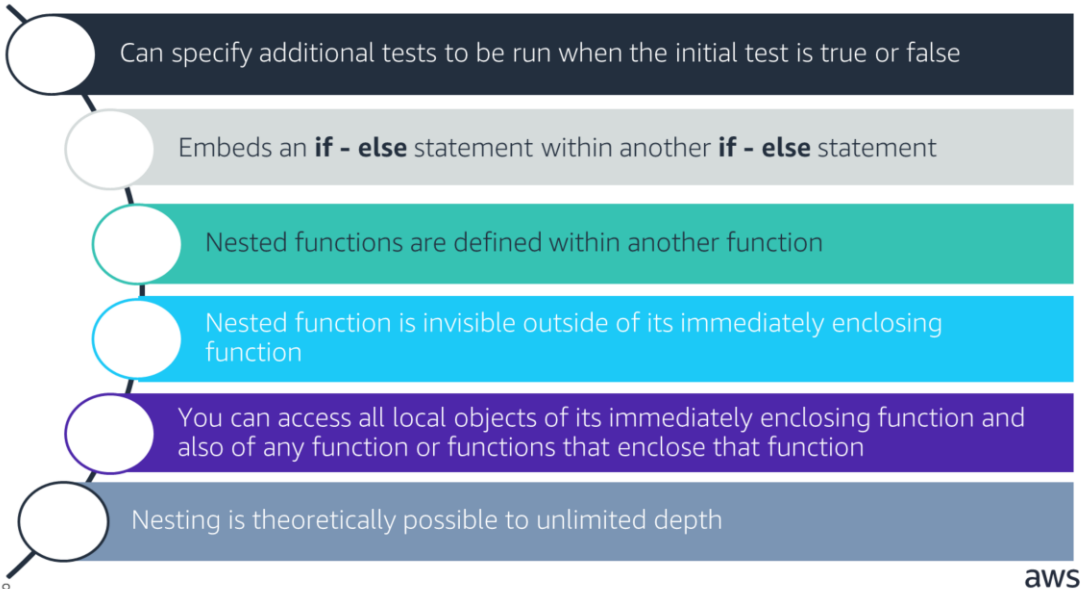
else statement

```
1 #!/bin/bash
2 # Copy file1 to /tmp
3 # Delete file1 if the copy was successful
4
5 if cp -f file1 /tmp
6 then
7 rm -f file1
8 else
9 echo "No such file."
10 fi
```

The **else** statement is invoked because **file1** does not exist

```
[root@server00 scripts]# ./delete.sh
cp: cannot stat ‘file1’: No such file or directory
No such file.
```

The **if - elif - else** statement



The **test** command

Part of conditional running of commands

Conditions are set, and then the test *exits* with a **0** for true and a **1** for false

Syntax: **test *<EXPRESSION>***

Testing whether this comparison is true or false.

```
1 #!/bin/bash
2 # comparing two values
3
4 test 100 -lt 99 && echo "Yes" || echo "No"
```

The comparison is false.

```
[root@server00 scripts]# ./comparison.sh
No
```

Integer comparison operators

Comparison operators compare two variables or quantities.

-eq is equal to
- **if ["$a" -eq "$b"]**

-ne is not equal to
- **if ["$a" -ne "$b"]**

-gt is greater than
- **if ["$a" -gt "$b"]**

-ge is greater than or equal to
- **if ["$a" -ge "$b"]**

-lt is less than
- **if ["$a" -lt "$b"]**

-le is less than or equal to
- **if ["$a" -le "$b"]**

< is less than (within double parentheses)
- **(("$a" < "$b"))**

<= is less than or equal to (within double parentheses)
- **(("$a" <= "$b"))**

> is greater than (within double parentheses)
- **(("$a" > "$b"))**

>= is greater than or equal to (within double parentheses)
- **(("$a" >= "$b"))**

String comparison operations

= or **==** is equal to
- **if ["$a" = "$b"]**
- **if ["$a" == "$b"]**

!= is not equal to
- **if ["$a" != "$b"]**
- This operator uses pattern matching within a [[...]] construct.

< is less than, in ASCII alphabetical order
- **if [["$a" < "$b"]]**
- **if ["$a" \< "$b"]**
- Note that the **<** must be escaped in a **[ ]** construct.

> is greater than, in ASCII alphabetical order
- **if [["$a" > "$b"]]**
- **if ["$a" \> "$b"]**
- Note that the **>** must be escaped in a **[ ]** construct.

-z string is null (that is, it has zero length)

-n string is not *null*

The **read** command

Reads user input into a script

Sets input as a variable

Sets the user input as a variable

Calls the variable and uses it as output

```
1 #!/bin/bash
2 # Greeting
3
4 echo "Hello. What is your name?"
5 read VARNAME
6 echo "Glad to meet you, $VARNAME."
```

```
[root@server00 scripts]# ./greeting.sh
Hello. What is your name?
Instructor
Glad to meet you, Instructor.
```

Loop statements

- Sections of a script can be configured to repeat themselves
- The loop can end:
  - After a specific number of repeats
  - Or until a condition is met
  - Or while a condition is true
- Looping extends power and complexity of scripts

Loop control statements

Used with loops to better manage their conditions

break – Stop running the entire loop

continue – If a condition is met, break out of loop

Example: If a specified value is discovered, then stop looping and continue through the script

Used in nested loops to continue to the next part of loop

The **for** statement

Loops the command a specified number of times

Bracketed by **do** and **done**

The **for** statement →

```
1 #!/bin/bash
2 # The for Loop
3
4 for x in 1 2 3 4 5 a b c d
5 do
6         echo "The value is $x"
7 done
```

Results →

```
[root@server00 scripts]# ./forloop.sh
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is a
The value is b
The value is c
The value is d
```

The **while** statement

Continues running the script as long as the specified condition is true

Bracketed by **do** and **done**

while statement

```
1 #!/bin/bash
2 # The while Loop
3
4 counter=1
5 while [ $counter -le 10 ]
6 do
7         echo $counter
8         ((counter++))
9 done
10
11 echo "Complete"
```

Result

```
[root@server00 scripts]# ./whileloop.sh
1
2
3
4
5
6
7
8
9
10
Complete
```

The **until** statement

Runs code until a condition becomes true

Bracketed by **do** and **done**

Similar to the **while** statement

until statement

```
1 #!/bin/bash
2 # The until Loop
3
4 counter=1
5 until [ $counter -gt 10 ]
6 do
7         echo $counter
8         ((counter++))
9 done
10
11 echo "Complete"
```

Result

```
[root@server00 scripts]# ./untilloop.sh
1
2
3
4
5
6
7
8
9
10
Complete
```

The **true** and **false** commands

- **true** and **false** commands are used with loops to manage their conditions
- Returns predetermined exit status (either a status of true or a status of false)
- A *Boolean expression* is used as an expression that produces a **** value when it is evaluated to be true or false
- Individual conditions can be created

```
1 #!/bin/bash
2 # The true Command
3
4
5 while true
6 do
7         echo `date`
8 done
```

Generates an infinite loop

Functions

- Small chunk of code that can be reused in a script
- Script within a script
- Scripts are more tidy, which makes them easier to understand and troubleshoot
- You don't need to rewrite the same code multiple times
- User-defined function
  - A function that the user of a program or environment provides
  - The context has the usual assumption is that functions are built into a program or environment
  - It is used to break down functions into manageable pieces that are easier to debug and monitor
- System-defined function
  - A function that is built into and run by the system
  - **Mailx:** A utility program for sending and receiving mail

Arguments to functions

- Arguments immediately follow the function name
- Everything between **{ }** is code that makes up a function
- Reference arguments in a function by **$1**, **$2**, . . .

Adding arguments

```
1 #!/bin/bash
2 # The add function
3
4 function add
5 {
6   x=$(( $1 + $2 ))
7   echo $x
8 }
9
10 add 99 1
```

Results

```
[root@server00 scripts]# ./function.sh
100
```

Return values of functions

Return value

```
1 #!/bin/bash
2 # Return Values
3
4 function demofunc()
5 {
6       myresult='42'
7 }
8
9 demofunc
10 echo $myresult
```

Result

```
[root@server00 scripts]# ./return.sh
42
```


aws re/start

Demonstration: Bash Shell Scripts

Your future with the company depends on your shell scripting abilities. Create a shell script that performs the following actions:

1. Automate the backup process.
2. Transfer the file to Information Assurance by using **scp**.

The following example is one possible Bash shell script for backing up a home directory. The basis of this example is that you want to back up the user's home directory. You want to save it to a directory on the remote server in /home/<*USER_NAME*>/backup, where <*USER_NAME*> is the user who is running the script. To ensure that you do not overwrite pre-existing backups, the .tar file name incorporates the current date.

```
#!/bin/bash

#begin by getting the user's user name and store in the username
variable
echo "Enter your username for the backup server:"
read username

#use scp to get the remote host where files will be backed up to
#and store in the remotehost variable.
echo "Enter the name of your remote backup server:"
read remotehost

#To keep from overwriting previous backups, the current date
#will be used to name the file.
backupfilename=`date +"%m-%d-%y"-`backup.tar

#On the remote server, it is assumed that each user will have
```

```
their
#own home directory, which is a mirror of their Linux home
directory.
#You will use the $HOME variable to capture that information.
homedirectory=$HOME

#The backup location is a concatenation of the remote host,
#the value of $HOME, and finally a directory that is called
backups.
backuplocation=$remotehost:/$homedirectory/backups

#Finally, run the commands.
tar -cvf $backupfilename `pwd`
scp $backupfilename $username@$backuplocation
```

Checkpoint questions

1. What are some tasks that you can automate by using shell scripts?
2. In what scenarios could you use shell scripts with conditional statements?

1. Some common tasks for shell scripts include:
    - Backing up important information
    - Moving files to storage locations so that only the newest files are visible
    - Finding duplicate files where thousands of files exist
2. Conditional statements can be useful when:
    - The script asks the user a question that has only a few answer choices
    - Deciding whether the script must be run
    - Ensuring that a command ran correctly, and taking action if it failed

Key takeaways

- Bash shell scripts offer a way to automate complex, multi-command operations into a single file.
- Using a script template helps ensure proper documentation in scripts.
- Running a script requires proper permissions.
- Variables that the script uses can be supplied from the command line or interactively from the user who is running the script.
- Conditional statements allow for different logic paths in the script (by using **if**, **gt**, equality, and others).
- *Security!* Ensure that the script contains only the required functionality.
- *Test!* Test all scripts to ensure that they function as expected.