



Bash Shell Scripts

Linux Fundamentals

© 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Welcome to Bash Shell Scripts.

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
- Run a shell script

In this module, 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
- Run a shell script



What is a script?

© 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

This section defines a script and covers common tasks that you can accomplish by using shell scripts.

What are scripts?

- Scripts are text files of commands and related data.
- When the text file is processed, the commands are run.
- Scripts can be set as scheduled tasks by using cron.
- Automation allows scripts to run more quickly than if they are run manually.
- Scripts are consistent due to automation removing the potential for manual errors.

Common script tasks:

- Creating backup jobs
- Archiving log files
- Configuring systems and services
- Simplifying repetitive task
- Automating tasks

Example script

An example backup script

```
#!/bin/bash
# Script to backup the home directory

tar -cf backup-home.tar /home/ec2-user

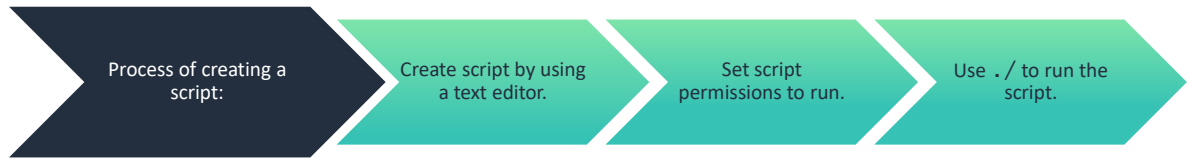
echo "backup job complete at `date`"
```

Script results

```
[ec2-user]$ ./backup.sh
tar: Removing leading `/' from member names
tar: /home/ec2-user/backup-home.tar: file is the archive; not dumped
"backup job complete at Fri Jun 18 08:13:30 UTC 2021"
[ec2-user]$ ls
backup-home.tar  displayName.sh  guess.sh  scripts  whilebreak
```

This script creates a tar archive with the content of the folder /home/ec2-user.

Shell scripts



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

Write the script in the text editor.

```
[ec2-user]$ ./hello.sh  
Hello ec2-user  
Today's date is : Fri Jun 18 08:22:27 UTC 2021  
[ec2-user]$
```

Run the script.

The process of creating a script follows these steps:

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

Amazon EC2 user data script

- Amazon Elastic Compute Cloud (Amazon EC2) is a virtual compute service.
- Shell scripts can run at creation time to install software on an EC2 instance.



This screen capture is an example of a user shell script that runs when the EC2 instance is created.

An HTTP web server is installed and run.

A basic HTML page is created.

The web server can be accessed via the public IP address of the EC2 instance.

Basic scripting syntax

© 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

This section introduces basic commands that are frequently included in shell scripts.

The # character

- Bash ignores lines that are preceded with #.
- The # character is used to define comments or notes to the user that might provide instructions or options.

```
[ec2-user]$ cat script.sh
#This is a comment
#echo "this line will be ignored"
echo "This line will print a message"
[ec2-user]$ ./script.sh
This line will print a message
[ec2-user]$
```

The second echo command runs and displays a message.

Only the second echo command runs and displays a message.

#!/bin/bash and #comments

- `#!` is referred to as a *shebang*.
- The first line defines the interpreter to use (it gives the path and name of the interpreter).
- Scripts must begin with the directive for which shell will run them.
- The 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.

```
[ec2-user]$ cat hello.sh
#!/bin/bash

echo "Hello `whoami`"
echo "Today's date is : `date`"
[ec2-user]$
```

If the first line does not define the interpreter, the operating system will find one, usually the one defined for the current shell.

Script documentation

- Some administrators create a script template, which contains all the relevant information and sections.
- The template might include the following:
 - Title
 - Purpose
 - Author's name and contact information
 - Special instructions or examples

```
#!/bin/bash
#.....
#Author : Jane Doe
# Date : 06/15/2021
#Description :Here is how you can document a script
#
#Usage :
# ./myScript.sh param1 [param2]
#param 1:
#param2:
#.....
#Version: 2.0.1
#Declared variables
#.....
#Script body
```

An example of script documentation

There is no exact format definition that scripts should follow.

Declare a Bash variable

- Declare (create) your own variables.
- Use these custom variables in scripts.

```
[ec2-user]$ cat displayName.sh
#!/bin/bash
NAME="Mary Major"
echo $NAME
[ec2-user]$ ./displayName.sh
Mary Major
[ec2-user]$
```

A variable called NAME has been created. The variable is set to the name Mary Major.

In the example, a variable NAME has been created.

Useful commands

Command	Description
echo	Displays information on the console
read	Reads a user input
subStr	Gets the substring of a string
+	Adds two numbers or combine strings
file	Opens a file
mkdir	Creates a directory
cp	Copies files
mv	Moves or renames files
chmod	Sets permissions on files
rm	Deletes files, folders, etc.
ls	Lists directories

This table lists the most useful commands that you can use in a script.

You can use all the shell commands that you saw earlier in the course, such as **grep**, **touch**, and **redirectors** (**>**, **>>**, **<**, **<<**).

You can also use the shell script to create programs using statements such as **if**, **else if**, **for**, **while**, **case**, and create functions using the **function** keyword.

Demonstration: Declaring a Variable in a Script



Overview

- Some variables are configured by default, but you can also declare your own variables. This functionality is often used when you write and run scripts. In this activity, you write a simple script to declare variables and use those variables in a practical use case.
- In this demonstration, a simple Bash script is created to demonstrate storing a user's name.

1. For this demonstration, create a file and enter the following text:

```
#!/bin/bash  
echo "what is your name?"  
read name  
echo "Hello $name"
```
2. As soon as you finish, save the file and run the following command (for this example, the file is named *getname.sh*):

```
chmod +x getname.sh
```
3. Run the script by entering the following (for this example, the file is named *getname.sh*):

```
./getname.sh
```

Operators

The = is used to assign the variable to a string.

The \$ is used to evaluate a variable.

```
#!/bin/bash  
sum=$(( $1 + $2 ))  
echo $1 + $2 equals $sum
```

The + is used as a math operator.

\$1 and \$2 are the parameters passed to the scripts.

You can also use the + operator to concatenate string.

Arguments

- Arguments are values that you want the script to act on.
- Arguments are passed to the script by including them in a script invocation command separated by spaces.
- For example, \$1 is the first argument, and \$2 is the second argument.

```
#!/bin/bash  
  
sum=$(( $1+$2 ))  
echo $sum
```

```
[ec2-user]$ ./math.sh 3 8  
11  
[ec2-user]$
```

In this example, the arguments input by the user are 3 and 8.

Therefore, \$1 is equal to 3 and \$2 is equal to 8.

Expressions

Use the `$((expression))` notation to evaluate an arithmetic expression in a bash shell script.

The value of the expression is assigned to the variable `sum`.



```
#!/bin/bash
sum=$(( $1 + $2 ))
echo $1 + $2 equals $sum
```

Expressions are evaluated and usually assigned to a variable for reference later in the script.

In this example, the script:

1. Evaluates the sum of the first two arguments passed to the script.
2. Assigns the result to the **sum** variable.
3. Echoes a message showing the value of the arguments and their computed sum.

Conditional statements

Conditional statements allow for different courses of action for a script depending on the success or failure of a test.

In the following script, the `rm` command is run only if the `cp` command is successfully completed.

```
#!/bin/bash
# Copy file1 from the current directory to /tmp.

cp file1 /tmp

# Delete file1 from the current directory
# if the copy was successful.

if [ $? -eq 0 ]
then
    rm file1
fi
```

Condition

```
[ec2-user]$ ls
delete.sh  displayName.sh  error.log  example.txt
Desktop    Documents      example.sh  file1
[ec2-user]$ ./delete.sh
[ec2-user]$ ls
delete.sh  displayName.sh  error.log  example.txt
Desktop    Documents      example.sh  hello.sh
[ec2-user]$ ls /tmp
file1
[ec2-user]$
```

Outcome: file1 does not exist anymore.

A conditional statement runs a command or block of commands only if a specified condition is satisfied. In the example shown, a script named **delete.sh** copies a file named **file1** from the current directory to the **/tmp** directory. If the copy command is successful, the script deletes the original file from the current directory.

Notice how the script uses the special variable **\$?** to test whether the copy command was successful or not. This variable contains a value that represents the exit status code of the last command that was run. If the command was successful, the return value is 0; otherwise, the return value is 1.

Logical control statements

© 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

The section focuses on basic logical control statements that are frequently included in shell scripts, such as `if`, `if-else`, `if-elif-else`, and `test`.

The `if` statement

- If the first command succeeds with an exit code of 0 (success), then the subsequent command runs.
- This is the simplest conditional statement.
- An `if` statement must end with the `fi` keyword.

```
#!/bin/bash
# Copy file1 to /tmp.
# Delete file1 if the copy was successful.

cp file1 /tmp

if [ $? -eq 0 ]
then
    rm file1
fi
```

An `if` statement is written as follows:

```
if <condition>
then
<command>
fi
```

Or if writing everything in one line:

```
if <condition>; then <command>; fi
```

Note that the semicolon (;) is required to separate multiple commands on the same line.

Indentation is used for better readability but is not required.

The if - else statement

- Defines two courses of action:
 - If the condition is true (then)
 - If the condition is false (else)

else statement

```
#!/bin/bash
# Copy file1 to /tmp.
# Delete file1 if the copy was successful.

cp file1 /tmp

if [ $? -eq 0 ]
then
    rm file1
else
    echo "No such file."
fi
```

The else statement is invoked because
file1 does not exist.

```
[ec2-user]$ ./delete.sh
cp: cannot stat 'file1': No such file or directory
No such file.
[ec2-user]$
```

An if-else statement is written as follows:

```
if <condition>
then
<command>
else
<other command>
fi
```

The `if - elif - else` statement

- Defines three courses of action:
 - If the condition is true (then)
 - Else if the condition is false but another condition is true (then)
 - If all of the previous conditions are false (else)

```
#!/bin/bash
# Compares $1 and $2

if [ $1 -gt $2 ]
then
    echo "the first number is greater then the second number"
elif [ $1 -lt $2 ]
then
    echo "the second number is greater then the first number"
else
    echo "the two numbers are equal"
fi
```

The example compares two numbers passed as parameters to the script:

- **-gt** means greater than
- **-lt** means lower than

An if-elif-else statement is written as follows:

```
if <condition>
then
<command>
elif <other condition>
then
<other command>
else
<default command>
fi
```

- You can embed if-elif-else statements.
- You can access all local objects of its immediately enclosing function and also of any function or functions that enclose that function.
- Nesting is theoretically possible to unlimited depth.

The `test` command

- Checks file types and compare values
- Conditions are tested, and then the test *exits* with a 0 for true and a 1 for false.
- Syntax: `test <EXPRESSION>`

Test whether this comparison is true or false.



```
#!/bin/bash
# Compares two values

test 100 -lt 99 && echo Yes || echo No
```

The comparison is false.



```
[ec2-user]$ ./compare.sh
No
[ec2-user]$
```

You use the `test` command to compare values.

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"))`

In these examples, you can see that brackets or double parentheses surround the condition.

- `((` are for numerical comparison:
 - `(($a < $b))` is equivalent to `[$a -lt $b]`
- You will also see some conditions using double brackets:
 - `if [[$a -lt $b]]`
 - This is basically a notation equivalent to a simple bracket with enhanced features, but this goes beyond this course.
- Note also that `"` are not mandatory: `if ["$a" -lt "$b"]` is equivalent to `if [$a -lt $b]`

Integer comparison operators examples

Comparison operators compare two variables or quantities.
The following tests are equivalent:

```
if [ "$1" -gt "$2" ]
then
    echo "... "
fi
```

```
if ((" $1" > "$2" ))
then
    echo "... "
fi
```

```
if [[ $1 -gt $2 ]]
then
    echo "... "
fi
```

```
if [ $1 -gt $2 ]
then
    echo "... "
fi
```

```
if (($1 > $2))
then
    echo "... "
fi
```

- Double quotes are not mandatory: **if ["\$a" -lt "\$b"]** is equivalent to **if [\$a -lt \$b]**
- The semicolon (;) is mandatory only if the clauses of the if statement are written in the same line: **if [\$a -lt \$b]; then ...; else ...; fi**

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*

Familiarize yourself with these string comparison operations. In the examples shown, `$a` and `$b` represent string variables.

String comparison operations example

```
#!/bin/bash
# Compares letters $1 and $2

if [ "$1" = "$2" ]
then
    echo letters are the same
elif [ $1 \< $2 ]
then
    echo the first letter is before the second letter
else
    echo the second letter is before the first letter
fi
```

```
[ec2-user]$ ./letters.sh a g
the first letters is before the second letter
[ec2-user]$ ./letters.sh a a
letters are the same
[ec2-user]$
```

Output

In a Bash shell script, the \$1 and \$2 variables contain the values of the first and second parameters passed to the script at runtime, respectively.

Loop statements

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



Loops provide you the ability to repeat sections of a script.

Loops can end:

- After a specific number of repeats (**for** statement)
- Until a condition is met (**until** statement)
- While a condition is true (**while** statement)

The for statement

- Loops the command a specified number of times
- Bracketed by do and done

```
#!/bin/bash
# The for loop

for x in 1 2 3 4 5 a b c d
do
    echo "the value is $x"
done
```

```
[ec2-user]$ ./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
[ec2-user]$
```

To loop a command a specific number of times, use the for statement.

The while statement

- Continues running the script as long as the specified condition is true
- Bracketed by **while** and **done**

```
#!/bin/bash
# The while loop

counter=1
while [ $counter -le 10 ]
do
    echo $counter
    ((counter++))
    if [ $counter = 11 ]
    then
        break
    fi
done
echo "loop exited"
echo "counter equals $counter"
```

```
[ec2-user] $ ./whileloop.sh 12
1
2
3
4
5
6
7
8
9
10
loop exited
counter equals 11
```

In this example, the while loop runs as long as the counter is lower or equal to 10, and the counter is not equal to the value of the first parameter passed to the script.

The body of the loop:

- Displays the value of the counter.
- Increments the counter by 1 (counter++ means counter = counter+1).
- Tests if the counter is equal to the first parameter value and breaks (exits) the loop if this is true.

The `until` statement

- Similar to the `while` statement
- Runs code until a condition becomes true
- Bracketed by `until` and `done`

```
#!/bin/bash
# The until loop

counter=1
until [ $counter -gt 10 ]
do
    echo $counter
    ((counter++))
done
echo "loop exited"
echo "counter equals $counter"
```

```
[ec2-user]$ ./untilloop.sh
1
2
3
4
5
6
7
8
9
10
loop exited
counter equals 11
[ec2-user]$
```

The `until` loop works the same way as the `while` loop.

In the example, the loop loops until the counter reaches 10 and then exits.

Loop control statement: Break

- Used with loops to better manage their conditions:
 - `break`: Stop running the entire loop.
 - `continue`: If a condition is met, break out of the current iteration but continue the 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.

```
[ec2-user]$ ./whilebreakloop.sh 5
1
2
3
4
loop exited
counter equals 5
[ec2-user]$
```

```
#!/bin/bash
# The while loop

counter=1
while [ $counter -le 10 ]
do
    echo $counter
    ((counter++))
    if [ $counter = 5 ]
    then
        break
    fi
done
echo "loop exited"
echo "counter equals $counter"
```

In this example, the while loop exits before the condition of the loop is met because the counter reaches the value passed as a parameter. Specifically:

- The parameter (\$1) is 5.
- The **if** condition is met when counter = 5.
- When this condition is true, the **break** statement is run, and the the loop exits.

Loop control statement: Continue

```
#!/bin/bash
# The continue statement

counter=1
while [ $counter -le 10 ]
do
    echo $counter
    if [ $counter = 7 ]
    then
        ((counter++))
        echo "Skipping the rest of the loop"
        continue
    fi
    ((counter++))
    echo "Completed all loop commands"
done
echo "loop exited"
echo "counter equals $counter"
```

```
[ec2-user]$ ./whileContinueLoop.sh 7
1
Completed all loop commands
2
Completed all loop commands
3
Completed all loop commands
4
Completed all loop commands
5
Completed all loop commands
6
Completed all loop commands
7
Skipping the rest of the loop
8
Completed all loop commands
9
Completed all loop commands
10
Completed all loop commands
loop exited
counter equals 11
[ec2-user]$
```

The **continue** keyword terminates the current loop iteration and returns control back to the top of the loop.

In the example, the bottom part of the loop is skipped when the counter is equal to 7. Specifically:

- The parameter is 7.
- The **if** condition is met when counter = 7.
- When this condition is true, the **continue** statement is run, and the rest of the loop is skipped.
- The loop keeps running until counter = 11 (original while condition).

The read command

Reads user input into a variable in the script.

Sets the user input as a variable →

```
#!/bin/bash
echo "Hello. what is your name?"
read VARNAME
echo "Glad to meet you, $VARNAME"
```

Calls the variable and uses it as output →

```
[ec2-user]$ ./welcome.sh
Hello. What is your name?
John
Glad to meet you, John
[ec2-user]$
```

Read incorporates the user's input as a variable into a script.

The true and false commands

- `true` and `false` commands are used with loops to manage their conditions.
- These commands return predetermined exit status (either a status of `true` or a status of `false`).
- A *Boolean expression* is an expression that produces a value of true or false when it is evaluated.

```
#!/bin/bash
while true
do
    echo "Enter a number between 1 and 10:"
    read val
    if [ $val -eq 7 ]
    then
        echo "You guessed right!"
        break
    fi
    echo "Wrong guess!"
done
```

```
[ec2-user]$ ./guessNumber.sh
Enter a number between 1 and 10:
3
Wrong guess!
Enter a number between 1 and 10:
6
Wrong guess!
Enter a number between 1 and 10:
9
Wrong guess!
Enter a number between 1 and 10:
5
Wrong guess!
Enter a number between 1 and 10:
7
You guessed right!
[ec2-user]$
```

In the example, a **while** statement with a **true** condition creates a continual loop that asks the user to enter a value between 1 and 10. If the user does not enter the value of 7, the loop continues; otherwise, the loop exists using a **break** statement.

The `exit` command

- Causes script to stop running and exit to the shell.
- Useful in testing
- Can return code status. Each code can be associated to a specific error
- For example:
 - `exit 0`: The program has completed without any error.
 - `exit 1`: The program has an error.
 - `exit n`: The program has a specific error.
- `$?` is a command to get the processing status of the last command that ran.

```
#!/bin/bash
touch myfile.txt
if [ $? -eq 0 ]
then
    echo "File created"
    exit 0
else
    echo "Error encountered when creating the file"
    exit 1
fi
```

```
[ec2-user]$ ./exit.sh
File created
[ec2-user]$
```

Successful exit

```
[ec2-user]$ ./exit.sh
touch: my/file.txt: No such file or directory
Error encountered when creating the file
[ec2-user]$
```

Error exit

aws re/start


This script creates a file using the **touch** command. It tests the status code of the **touch** command and exits accordingly. The slide shows the output of a successful exit and of an error exit. The error exit is caused by running the script with an invalid file name in the touch command. In this case, the name included an invalid forward slash character.

Command substitution (review)

- Commands can be placed in the syntax of other commands.
- Commands are surrounded by backticks (` `).
- Commands can be useful in scripts.

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

Command substitution



Command substitutions are useful in scripts.



Run a shell script

© 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

In this section, you run a shell script.

Script permissions (review)

- Script files must have run permissions.
- `chmod 744 hello.sh` permits the user to run the script but not a group or other users.
- `chmod u-w hello.sh` removes the write permission on the script for the user.

```
[ec2-user]$ ls -al hello.sh
-rwxr-xr-x 1 ec2-user ec2-user 67 Jun 15 12:44 hello.sh
[ec2-user]$ chmod 744 hello.sh
[ec2-user]$ ls -al hello.sh
-rwxr--r-- 1 ec2-user ec2-user 67 Jun 15 12:44 hello.sh
[ec2-user]$ chmod u-w hello.sh
[ec2-user]$ ls -al hello.sh
-r-xr--r-- 1 ec2-user ec2-user 67 Jun 15 12:44 hello.sh
[ec2-user]$
```

chmod 744 is:

- Read (r) for the owner, the group of the owner, and others
- Write (w) for the owner only
- Execute (x) for the owner only
- Hence `rwxr--r--`

The second command, **chmod u-w**, removes the write right for the owner:

`rwxr--r--` becomes `r-xr--r--`

Running a script in Bash

- Bash searches for executables by using the `$PATH` variable. Bash assumes that any runnable command or script will be along that path.
- If the script is not along the path, then precede the executable name with `./`
- For example, in your home directory, you wrote a script called **myscript.sh** and you want to run it:
 - `myscript.sh` fails because Bash does not check your home directory for executables.
 - `./myscript.sh` succeeds.
- Be sure to update the settings to persist beyond the current session.
- Review: A user's profile is loaded from the files that are stored in the user's home directory:
 - `/home/username/.bashrc`
 - `/home/username/.bash_history`
 - `/home/username/.bash_profile`

Review this information so that when you run a script in Bash you don't have any issues.

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.

```
[ec2-user]$ hello.sh
-bash: hello.sh: command not found
[ec2-user]$ ./hello.sh
Hello ec2-user
Today's date is : Fri Jun 18 07:43:55 UTC 2021
[ec2-user]$ /home/ec2-user/hello.sh
Hello ec2-user
Today's date is : Fri Jun 18 07:44:12 UTC 2021
[ec2-user]$
```

The home folder is not in \$PATH

- **hello.sh** fails.
- **./hello.sh** succeeds because the current folder is the folder containing the script.
- Using the full path **/home/ec2-user/hello.sh** works from any folder.

Checkpoint questions



What are some tasks that you can automate by using shell scripts?



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. Shell scripts with conditional statements can be used 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.
- By using a script template, you can help 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 create different logic paths in the script (by using `if`, `gt`, `equality`, and others).
- **Security!** Check that the script contains only the required functionality.
- **Test!** Test all scripts to confirm that they function as expected.

Some key takeaways from this lesson include:

- Bash shell scripts offer a way to automate complex, multi-command operations into a single file.
- By using a script template, you can help 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 create for different logic paths in the script (by using **if**, **gt**, `equality`, and others).
- **Security!** Check that the script contains only the required functionality.
- **Test!** Test all scripts to confirm that they function as expected.



Thank you

© 2021 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited. Corrections, feedback, or other questions? Contact us at <https://support.aws.amazon.com/#/contacts/aws-training>. All trademarks are the property of their owners.



Thank you.