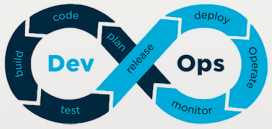
DolfinED

Transforming Lives Through Learning



Bash Shell Scripting Mastery In Arabic

By Mena Magdy Halem

Course Introduction

DolfinED
Transforming Lives Through Learning

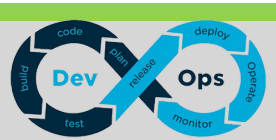
Why do you need to learn Bash Scripting?

- Here are several reasons why bash scripting is important:
  - Automation of Repetitive Tasks
  - Task Scheduling
  - File and Text Processing
  - Cloud Automation
  - Continuous Integration/Continuous Deployment (CI/CD)

- Target Audience
  - DevOps Engineer
  - Cloud Engineer
  - System Administrators
  - Linux Administration
  - Developers

- Prerequisite for Learning Bash Scripting:
  - Basic of linux

About The Course



Bash Shell Scripting Mastery

70+ Video

10+ Hours

30+ Video Demos

Hands-on Lab Guide

DolfinED
Transforming Lives Through Learning

Table Of Content

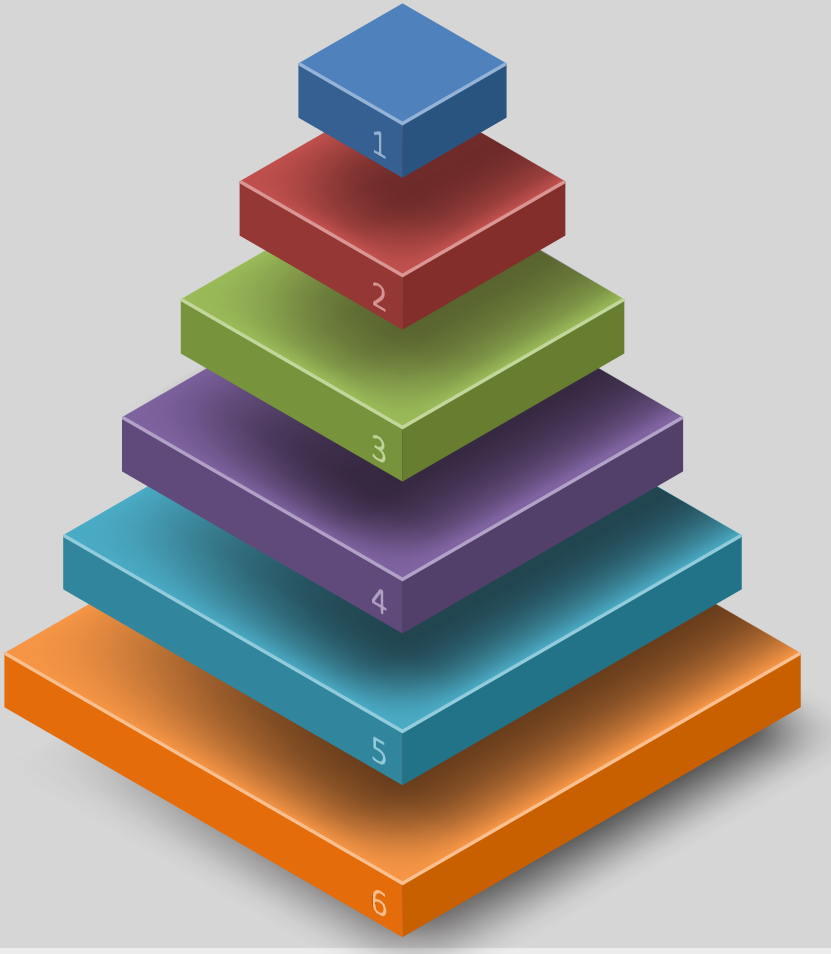
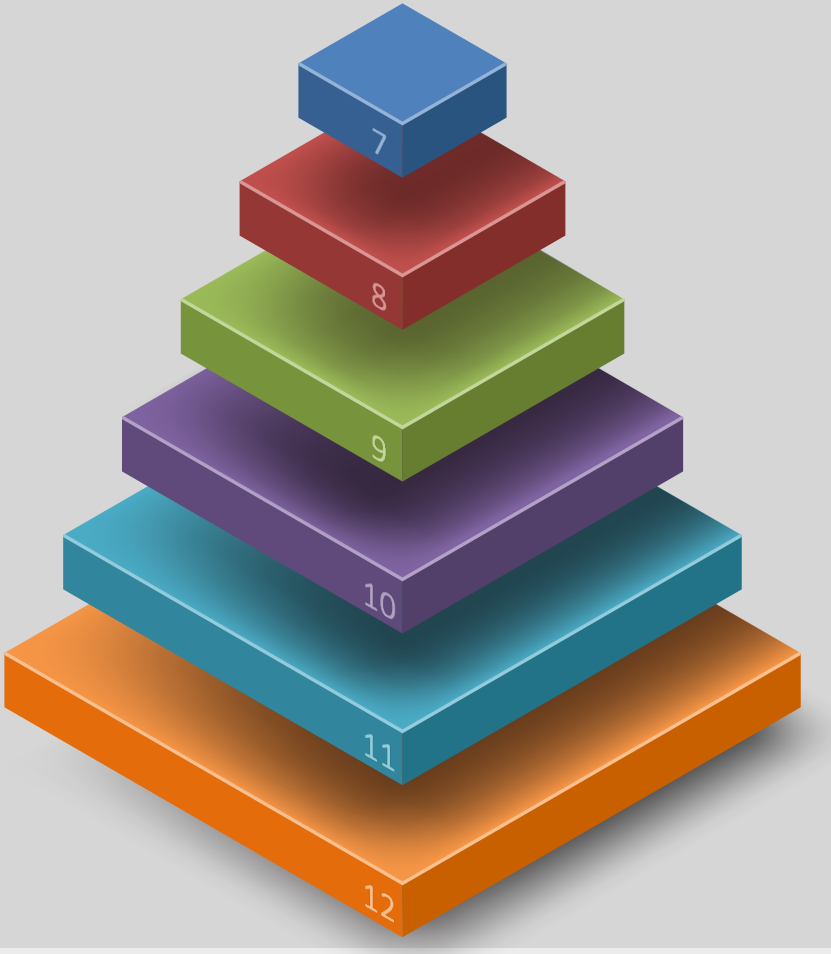


Table Of Content



Bash Scripting - Introduction

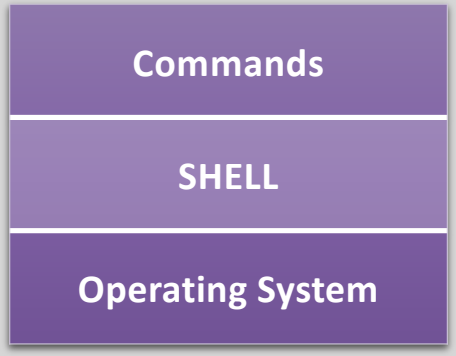
DolfinED
Transforming Lives Through Learning

What is Bash Scripting?

DolfinED
Transforming Lives Through Learning

What Is A Shell?

- A shell is a command-line interface (CLI) program that provides a user interface to an operating system's services.
- The shell acts as an intermediary between the user and the operating system
- It allows users to interact with the computer by entering commands, executing programs, and managing files and directories.

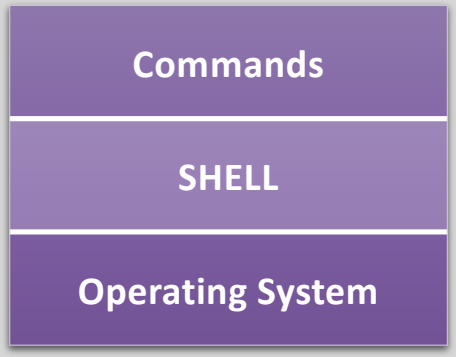


Common Shells

There are several types of shells available:

- Bash
- Zsh (Z Shell)
- Ksh (Korn Shell)
- Csh (C Shell)

Each shell has its own syntax and set of features, but they all share the common purpose of providing a command-line interface for interacting with the operating system.



What is BASH?

- Bash, short for "Bourne Again SHell“
- Bash is one of the most commonly used Unix/Linux shells and is the default shell in many Linux distributions.
- With Bash you can automate tasks using scripts.

What is Bash Scripting?

- Bash is a command-line interpreter
- A bash script is a series of commands written in a file.
  - The commands are read and executed by the bash program.
  - The program executes line by line.
- Bash scripting supports variables, conditional statements, loops, functions, and other programming constructs.

Basics of Scripting

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- How To Create a Shell Script File
- How To Run A Shell Script File
- Launch an AWS ubuntu EC2 Instance
- Demo - How To Run A Shell Script File

How To Create A Shell Script File

DolfinED
Transforming Lives Through Learning

How To Create A Shell Script File

- Writing a Bash script involves creating a text file with a sequence of commands and instructions to be executed in the Bash shell.
- Step 1: Choose a text editor: Start by selecting a text editor that you're comfortable with. Popular choices include Vim, Nano, or even a code editor like Visual Studio Code.
- Step 2: The first line of your script should begin with a shebang **(#!)** followed by the path to the Bash interpreter.
  - This line tells the system which interpreter to use to execute the script.
- Comments are lines that provide information about your script.
  - They are ignored by the interpreter but can be helpful for documentation and understanding.
- Step 3: Save the file with a .sh extension, such as script.sh, to indicate that it's a Bash script.

Script.sh

```
#!/bin/bash
# This is a Comment
```

How To Run A Shell Script File

DolfinED
Transforming Lives Through Learning

How To Run A Shell Script File

Step 1: You need to make the script file executable

- Use the 'chmod' command to set the executable permission on the script file.

Script.sh

```
#!/bin/bash
# This is a Comment
echo "Hello From DolfinED"
```

Step 2: Run the script

```
ubuntu@DolfinED:~$ ./script.sh
Hello From DolfinED
```

```
ubuntu@DolfinED:~$ ls
Script.sh
```

```
ubuntu@DolfinED:~$ bash ./script.sh
Hello From DolfinED
```

Chmod Calculator
https://chmod-calculator.com/

DolfinED
Transforming Lives Through Learning

Adding Scripts To Your PATH

- Determine the directory where your script is located.
  - For example, let's say your script is in the directory **/scripts**
- Open the shell configuration file ~/.profile , You can open the file using a text editor.
- Add a line to the configuration file that exports the directory containing your scripts to the PATH variable.
- Now, you can run your scripts from anywhere in the terminal without specifying the full path.
  - Just use the script name as a command, and the shell will find and execute it because it's in one of the directories listed in the PATH variable.

/.profile

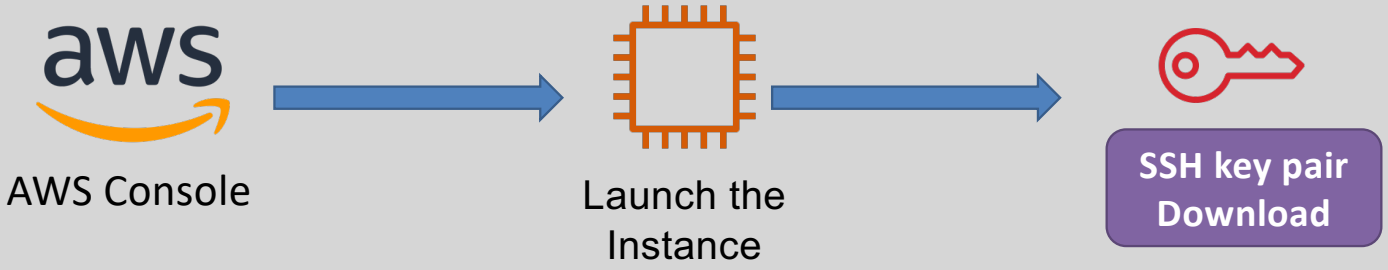
```
export PATH=$PATH:$HOME/scripts
```

Hands-on Labs (HoLs)

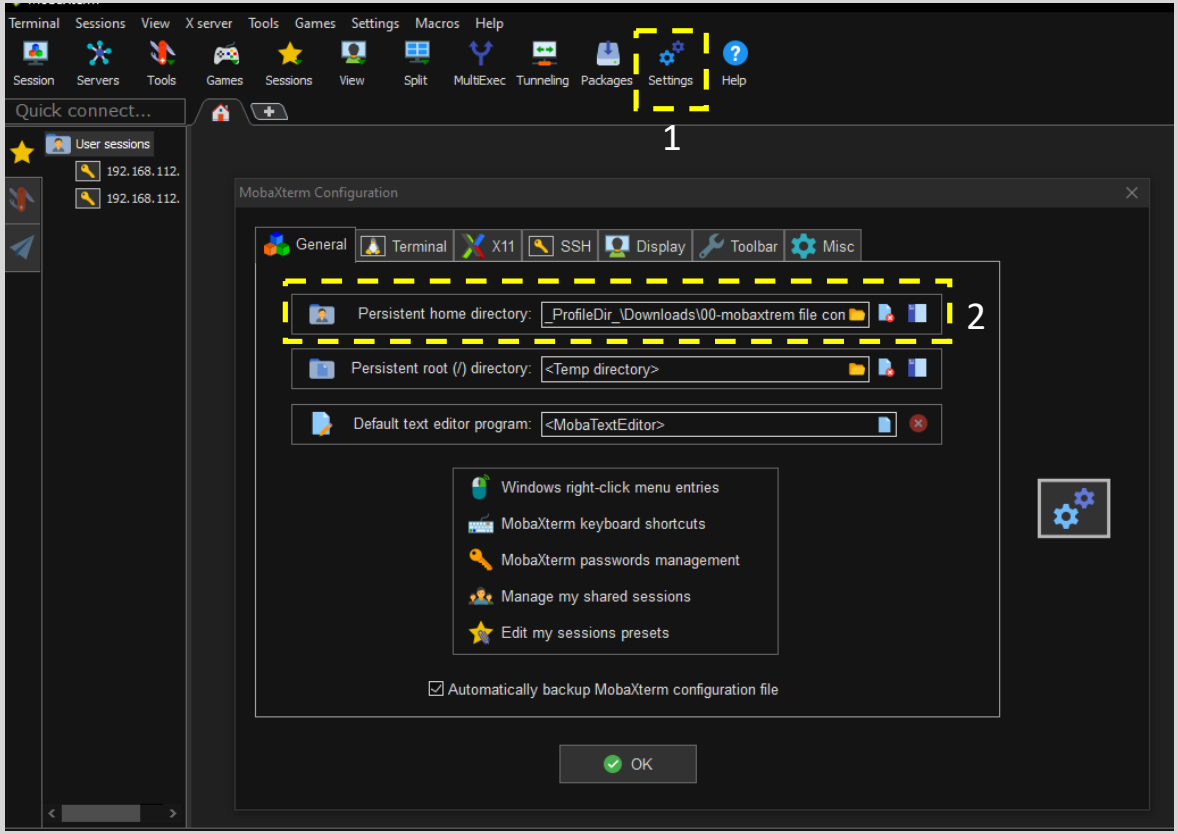
Launch an AWS ubuntu EC2 Instance

Launch an AWS ubuntu EC2 Instance - Steps

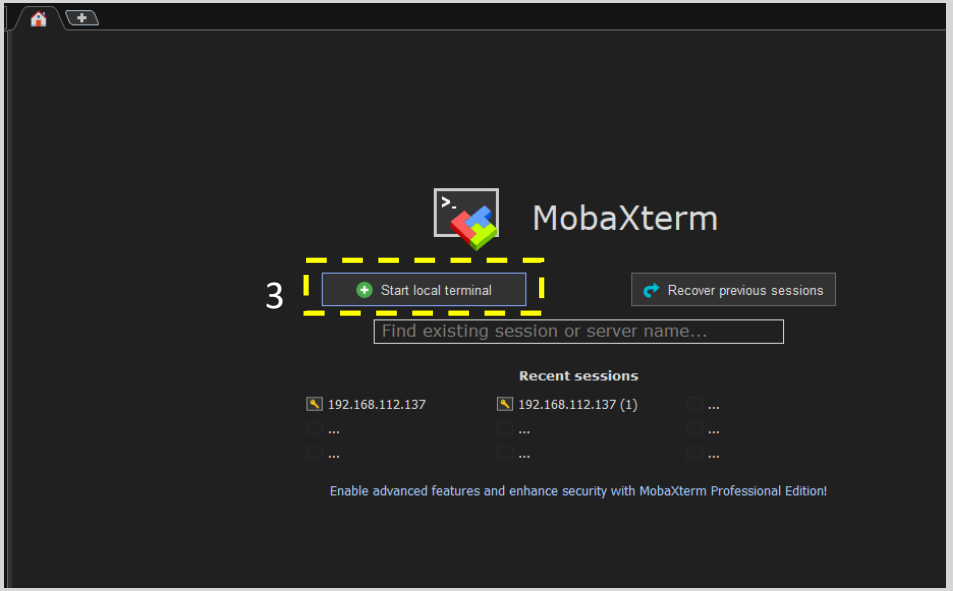
- Open the AWS console
- Launch a new EC2
- Create a new key pair (must download and save it)



Login to the EC2 instance using MobaXterm



Download MobXterm




```
ssh -i "k8s.pem" ubuntu@ec2-3-92-62-38.compute-1.amazonaws.com
```


DolfinED
Transforming Lives Through Learning

Hands-on Labs (HoLs)

How To Create And Run A Shell Script File

Variables In Bash

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- Local and Global Variables
- Environment Variables
- Exit Status
- Special Variables
- Arithmetic Operations
- Command Substitution
- Brace Expansion
- Reading User Input

Local Variables in Bash

DolfinED
Transforming Lives Through Learning

Variables In BASH

A variable is a named memory location which is a reference to a value that can be used throughout a script or in the interactive shell.

- Variables are used for storing values of different types during program execution.
- Based on the scope of the variable (from where it can be accessed) there are two types:
  - Local Variables
  - Global Variables

Local Variables

- A local variable is a special type of variable which has its scope (can only be used) within the script or shell in which it is defined.
- The syntax for a local variable: Variable_name =value
- A local variable name must start with a letter or underscore.
- Do not use special characters (such as @,#,%,$) in a local variable name
- Variable are case sensitive
- Allowed: VARIABLE, VAR1234able, var_name, _VAR
- Not allowed: 1var, %name, $myvar, var@NAME, myvar-1
- To read a variable, write $ sign and Variable_name

```
ubuntu@DolfinED:~$ name=DolfinED
```

```
ubuntu@DolfinED:~$ echo $name
DolfinED
```

Hands-on Labs (HoLs)

Local Variables in Bash

Environment Variables

DolfinED
Transforming Lives Through Learning

Environment Variables

- Environment variables are predefined variables that are available to the Bash shell.
- They are part of the shell's environment and can be accessed by any program running in that environment.
- Environment variables are system defined, they are typically set by the operating system, system utilities, or by the user.
- Bash provides several commonly used Linux environment variables, such as:
  - PATH: Specifies the directories in which the shell looks for executable programs.
  - HOME: Represents the path to the user's home directory.
  - USER: Stores the username of the currently logged-in user.
  - SHELL: Contains the path to the current shell interpreter (e.g., /bin/bash).
  - PWD: Represents the current working directory.

Environment Variables

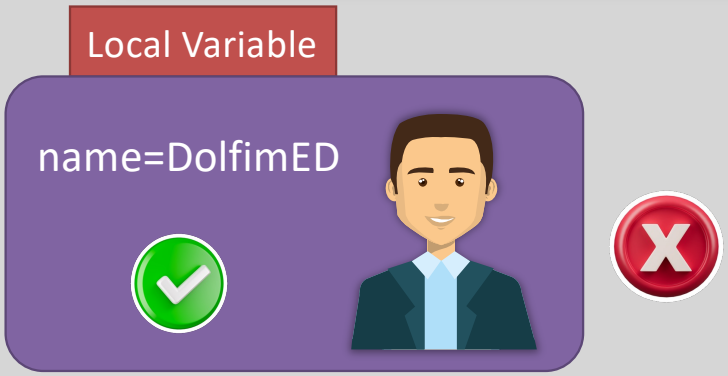
The system-defined Environment Variables can be viewed using the env or printenv commands.

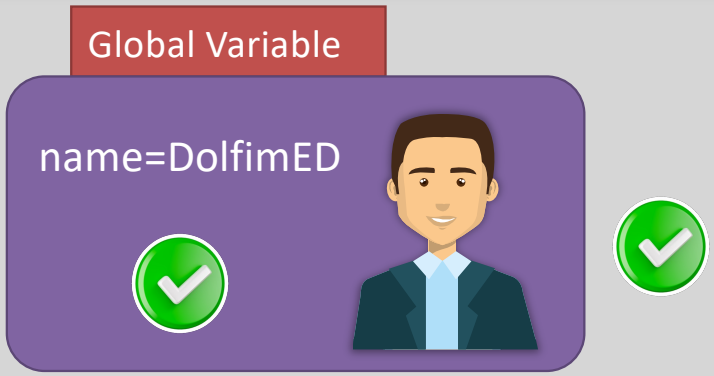
```
ubuntu@DolfinED:~$ env
SHELL=/bin/bash
PWD=/home/ubuntu
LOGNAME=ubuntu
XDG_SESSION_TYPE=tty
MOTD_SHOWN=pam
HOME=/home/ubuntu
.
.
.
```

Global Variables

Global variables are variables that are defined in the global scope, outside of any specific function or block of code.

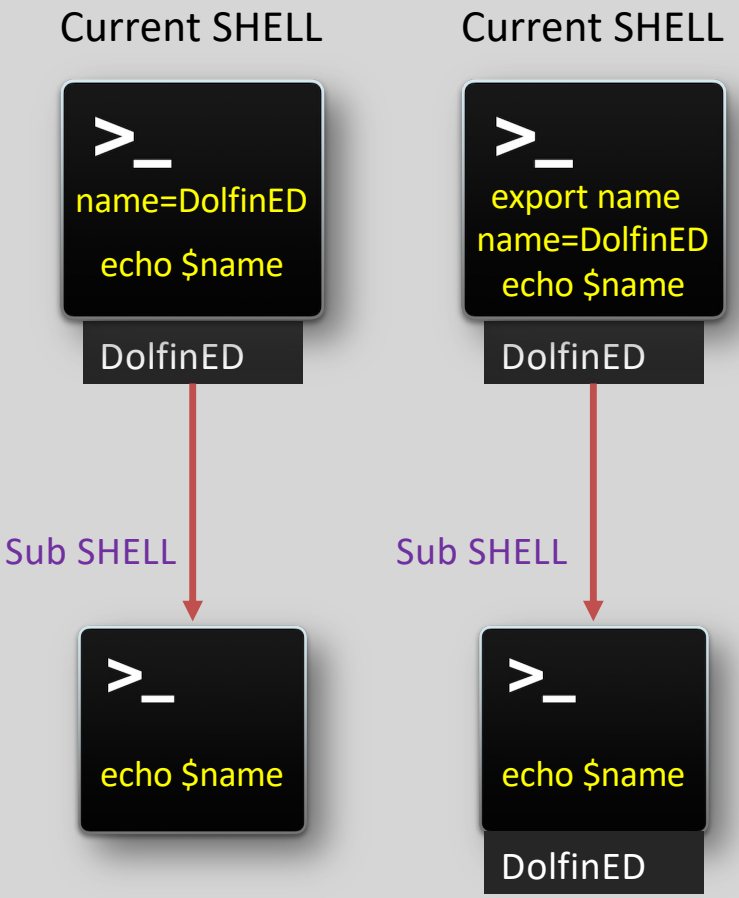
- Global variables can be used by all Bash scripts on your system
- Global variables are typically used for data that needs to be shared and accessed by multiple functions or throughout the entire program.
- They provide a simple way to share configuration settings, between multiple applications and processes in Linux.





Creating Environment Variables Using Bash Export Command

- By default, a child process or subshell does not inherit variables set in its parent shell.
- We can make a local variable available to any child processes of that shell using the built-in bash export command.
  - It will cause the variable to be inherited by subshells (child process).
  - This is basically creating a new Environment Variable local to this shell and its subshells.



DolfinED
Transforming Lives Through Learning

Quotes & Bash Variables

Quotation marks play an essential role in the bash shell.

- Depending on whether you use double quotes (“) or single quotes (‘), it can change the output.
  - Double quotes (“) :
    - It prints the actual variable value.
  - Single quotes (‘) :
    - It shows only the variable name instead.

```
ubuntu@DolfinED:~$ name=DolfinED
```

```
ubuntu@DolfinED:~$ echo “$name”
DolfinED
```

```
ubuntu@DolfinED:~$ echo ‘$name’
$name
```

Hands-on Labs (HoLs)

Environment Variables

Hands-on Labs (HoLs)

Environment Variables – Part 2

Exit Status

DolfinED
Transforming Lives Through Learning

Exit Status

The exit status is a numeric value that represents the outcome or result of the execution of a command or script.

- The exit status indicates whether the command or script completed successfully or encountered an error during execution.
- After executing a command or script, the exit status of the most recently executed command or script is stored in the special variable "$?"

```
ubuntu@DolfinED:~$ ls
sc1.sh  sc2.sh
```

```
ubuntu@DolfinED:~$ echo $?
0
```

Exit Status (cont.)

The exit status is represented by a numeric value between 0 and 255.

- 0 exit status means the command was successful without any errors.
- Any non-zero value indicates failure or an error.

Exit status	Description
0	The Command or script is successfully
1	Catchall for general errors
2	Misuse of shell built-ins, such as incorrect command-line arguments
126	Command cannot execute
127	Command not found

Hands-on Labs (HoLs)

Exit Status

Special (Pre-Defined) Variables In Bash

DolfinED
Transforming Lives Through Learning

Predefined Variables

- In Bash, special variables are those variables that are intentionally set by the shell internally so that it is available to the user.
- They provide access to information related to the script's execution, command-line arguments, and other useful data.

Predefined Variables	Description
$0	Gets the name of the current script
$#	Gets the number of arguments passed while executing the bash script.
$*	Gives you a string containing every command-line argument
$1-$9	Stores the first 9 arguments.
$!	Shows the process ID of the last background command.

```
ubuntu@DolfinED:~$ ./sc1.sh one two three
```

Hands-on Labs (HoLs)

Special (Pre-defined) Variables

Arithmetic Operations in Bash

DolfinED
Transforming Lives Through Learning

Arithmetic Operations In Bash

- Arithmetic in Bash scripting refers to the capability of performing mathematical calculations and operations within a script.
- Bash provides built-in arithmetic capabilities for integers and supports basic arithmetic operations like addition, subtraction, multiplication, division, and more.
- Arithmetic expansion is done using the $((...)) syntax, where an arithmetic expression is enclosed within double parentheses.
- Example: result=$((2 + 3)) assigns the value 5 to the variable result
- **let** is a built-in function of Bash that allows us to do simple arithmetic operations.

```
ubuntu@DolfinED:~$ result=$((2 + 3))
```

```
ubuntu@DolfinED:~$ echo $result
5
```

```
ubuntu@DolfinED:~$ let a=2+3
```

```
ubuntu@DolfinED:~$ echo $a
5
```

Arithmetic Operations (cont.)

Bash supports the following basic arithmetic operations:

Operator	Name
+	Addition
–	Subtraction
*	Multiplication
/	Division
var++	Increase the variable var by 1
var--	Decrease the variable var by 1

Comparison Operators

Bash also supports comparison operators for evaluating conditions in arithmetic operations

Operator	Name
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Hands-on Labs (HoLs)

Arithmetic Operations In Bash

Command Substitution

DolfinED
Transforming Lives Through Learning

Command Substitution

- Command substitution is a feature in Bash scripting that allows users to capture the output (result) of a command and use it as part of another command or store it in a variable for later use.
- Command substitution can be done using two syntax forms: $() (preferred) or backticks ``.
- Example: files=$(ls) captures the output of the ls command (list of files in the current directory) and stores it in the files variable.

```
ubuntu@DolfinED:~$ files=$(ls)
```

```
ubuntu@DolfinED:~$ files=`ls`
```

```
ubuntu@DolfinED:~$ echo $files
sc1.sh  sc2.sh
```

Hands-on Labs (HoLs)

Command Substitution

Brace Expansion

DolfinED
Transforming Lives Through Learning

Brace Expansion

- Brace expansion in Bash scripting is a feature that allows you to generate multiple strings or expand a sequence of values based on a pattern defined within curly braces {}
- It provides a concise way to generate a series of strings or perform string manipulation operations.
- By separating strings or values with commas inside curly braces, you can generate multiple strings or values.
- Brace expansion also allows you to expand a sequence of values based on a pattern using the {start..end} syntax.

```
ubuntu@DolfinED:~$ echo {one,two,three}
one two three
```

```
ubuntu@DolfinED:~$ echo {1..5}
1 2 3 4 5
```

Hands-on Labs (HoLs)

Brace Expansion

Reading User Input

DolfinED
Transforming Lives Through Learning

Read User Input

- In Bash scripting, you can read user input using the read command.
  - The command allows you to prompt the user for input and store the entered value in a variable.
- The basic syntax for the read command is read variable_name
- You can provide a prompt message to the user using the -p option.
- You can read sensitive information like passwords without displaying the user's input on the screen, use the -s option.
- You can set a timeout for the read command to automatically proceed if no input is received within a specified time using the -t option.

Hands-on Labs (HoLs)

Reading User Input

Conditions In Bash Scripting - If Statements

DolfinED
Transforming Lives Through Learning

If Statements

In this section, we will learn:

- Understanding If Statements
- logical operations
- Examples
- Learn How to use && and ||
- Wildcards
- Regular Expressions

Conditional Operators

DolfinED
Transforming Lives Through Learning

Conditional Operators

- Conditional Operators are used to perform logical and comparison operations to evaluate conditions.
- These operators allow you to make decisions and control the flow of your script based on the truth or falsehood of specific conditions.
- Here are the commonly used conditional operators in Bash:
  - Comparison Operators for Numerical Values
  - Comparison Operators for String Values
  - Logical Operators
  - File Operators

Comparison Operators for Numerical Values

Operator	Name	Example
-eq	Equal	["$a" -eq "$b"]
-ne	Not equal	["$a" -ne "$b"]
-gt	Greater than	["$a" -gt "$b"]
-lt	Less than	["$a" -lt "$b"]
-ge	Greater than or equal	["$a" -ge "$b"]
-le	Less than or equal	["$a" -le "$b"]

Comparison Operators for String Values

Operator	Name	Example
=	Equal	["$a" = "$b"]
!=	Not equal	["$a" != "$b"]
>	Greater than	["$a" > "$b"]
<	Less than	["$a" < "$b"]
-z	Empty string	[-z "$a"]

Logical Operators

Operator	Name	Example
&&	AND	[condition1] && [condition2]
||	OR	[condition1] || [condition2]
!	NOT	! [condition]

File Operators

Operator	Name	Example
-e	File exists	[-e "$file"]
-f	Regular file	[-f "$file"]
-d	Directory	[-d "$dir"]
-r	Readable	[-r "$file"]
-w	Writable	[-w "$file"]
-x	Executable	[-x "$file"]

Conditional Operators (cont.)

- There are two main forms of conditional expressions:
  - using double brackets [[..]] and
  - using single brackets [..]
- The double brackets [[..]] provide more advanced features, enhanced syntax, and improved handling of variables and strings
  - They are generally preferred over single brackets [..] for conditional expressions in Bash scripts

Hands-on Labs (HoLs)

Conditional Operators

Understanding If Statements

DolfinED
Transforming Lives Through Learning

Understanding If Statement

- The If statement is a fundamental control structure that allows you to make decisions in your script based on the evaluation of a condition.
- It enables you to execute different blocks of code depending on whether the condition is true or false.
- The basic syntax of the if statement is shown

Script.sh

```
#!/bin/bash
if [ condition ]
then
   # Code to execute
fi
```

Understanding If Statement (cont.)

- The condition can be a comparison between variables, a file test, a string comparison, or any expression that evaluates to true or false.
- Make sure to enclose the condition within square brackets [...].
- There should be spaces before and after the brackets.
- Use the then keyword to specify the block of code that will be executed if the condition is true.
- End the if statement with the fi keyword

Script.sh

```
#!/bin/bash
if [ condition ]
then
   # Code to execute
fi
```

If, else

Optionally, use the else keyword to specify the block of code that will be executed if the condition is false. The else block follows the same syntax as the then block.

Script.sh

```
#!/bin/bash
if [ condition ]
then
   # Code to execute
else
   # Code to execute
fi
```

If, elif – Multiple Conditions

- if and elif (short for "else if") are conditional statements that allow you to handle multiple conditions in your script.
- The if statement is used to start the conditional block.
- You provide a condition to evaluate inside the first set of square brackets [...].
  - If this condition evaluates to true, the code block after then will be executed.
- The elif statement, is used to check an additional condition if the previous if or elif condition is false.
- You can have multiple elif statements to handle multiple conditions.
- The fi statement marks the end of the if statement

Script.sh

```
#!/bin/bash
if [ condition1 ]
then
   # Code to execute
elif [ condition2 ]
then
   # Code to execute
else
   # Code to execute
fi
```

Hands-on Labs (HoLs)

Understanding If Statement

Hands-on Labs (HoLs)

If Statement Using File Operators And Logical Operators

Regular Expressions

DolfinED
Transforming Lives Through Learning

Regular Expressions

- Regular Expressions is a powerful tool for pattern matching and text manipulation.
- Regular expressions allow you to define search patterns using a combination of characters, metacharacters, and quantifiers, enabling you to perform complex and flexible matching operations on text data.
- Bash supports regular expressions through the [[...]] construct when used with the =~ operator. This operator allows you to compare a string against a regular expression pattern.

Script.sh

```
#!/bin/bash
if [[ $var =~  regular_expression ]]
then
   # Code to execute
else
   # Code to execute
fi
```

Regular Expressions (cont.)

Characters	Description
. (dot)	Matches any single character except a newline
*	Matches zero or more occurrences of the preceding character or group
^	Matches the start of a line
$	It matches the end of the string
?	It matches exactly one character in the string or stream
\	It is used for escape following character

Hands-on Labs (HoLs)

Regular Expressions

Case

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- Understanding Case Statement
- Demo - Case Statement

Understanding Case Statement

DolfinED
Transforming Lives Through Learning

Understanding Case Statement

- The case statement is a powerful control structure that allows you to perform multiple conditional checks on a single variable.
- The case keyword starts the case statement, indicating that you are going to check a variable against various patterns.
- Each pattern is followed by a right parenthesis) and a double semicolon ;;.
  - The double semicolon signifies the end of a code block associated with a specific pattern.

Script.sh

```
#!/bin/bash
case variable in
  pattern1)
    # Code to execute
    ;;
  pattern2)
    # Code to execute
    ;;
  pattern3)
    # Code to execute
    ;;
  *)
    # Code to execute
    ;;
esac
```

Understanding Case Statement (cont.)

- The * pattern at the end is optional and acts as a catch-all for any value that doesn't match any of the preceding patterns.
  - If the variable does not match any of the specified patterns, the code block following the * pattern will be executed.
- The esac keyword marks the end of the case statement.

Script.sh

```
#!/bin/bash
case variable in
   pattern1)
      # Code to execute
      ;;
   pattern2)
      # Code to execute
      ;;
   pattern3)
      # Code to execute
      ;;
   *)
      # Code to execute
      ;;
esac
```

Hands-on Labs (HoLs)

Case Statement

Array and for Loop

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- What is an Array?
- Demo - Array
- Explaining For Loop
- Demo – For Loop
- Continue And Break
- Select Command
- Demo - Select Command

What is an Array?

DolfinED
Transforming Lives Through Learning

What is an Array?

- An array is a data structure that can hold multiple values under a single variable name.
- Arrays allow you to store and manipulate a collection of elements, such as numbers, strings, or a mix of different data types.
- The syntax of creating an array: array_name=(element1 element2 ... elementN)

Syntax	Description
echo ${arr[2]}	Print the third elements
echo ${arr[@]}	Print all elements
echo ${#arr[@]}	Print the number of elements
arr+=(new)	Add element to an array
unset arry[2]	Remove the element from array that index ID 2

Hands-on Labs (HoLs)

Arrays

For Loop

DolfinED
Transforming Lives Through Learning

For Loop

- The for loop is a control structure that allows you to iterate over a list of items and execute a block of code for each item in the list.
- It is a fundamental construct used for repetitive tasks, such as processing elements of an array, iterating through files in a directory, or executing a command multiple times with different arguments.

Script.sh

```
#!/bin/bash
for variable in item1 item2 ... itemN
do
    # Code to be executed for each item
done
```

For Loop

- The for keyword marks the beginning of the for loop.
- variable is a user-defined variable that will take on the value of each item in the list as the loop iterates.
- in item1 item2 ... itemN specifies the list of items over which the for loop will iterate.
- The loop will execute the code block enclosed between do and done for each item in the list.

Script.sh

```
#!/bin/bash
for variable in item1 item2 ... itemN
do
   # Code to be executed for each item
done
```

Hands-on Labs (HoLs)

For Loop

Continue And Break

DolfinED
Transforming Lives Through Learning

Continue

Continue and Break are control statements that can be used within loops, including for loops, to control the flow of execution.

- Continue Statement:
  - The continue statement is used to skip the rest of the current iteration of a loop and continue with the next iteration.
  - When the continue statement is encountered, the remaining code block inside the loop for the current iteration is skipped, and the loop immediately moves to the next iteration.

Script.sh

```
#!/bin/bash
for i in {1..5}
do
   if [ $i -eq 3 ]; then
      continue
   fi
   echo “Number: $i"
done
```

```
Number: 1
Number: 2
Number: 4
Number: 5
```

Break Statement

The break statement is used to exit the loop prematurely when a specific condition is met.

- When the break statement is encountered, the loop terminates immediately, and the execution continues with the code following the loop

Script.sh

```
#!/bin/bash
for i in {1..5}
do
   if [ $i -eq 3 ]; then
      break
   fi
   echo “Number: $i"
done
```

```
Number: 1
Number: 2
```

Hands-on Labs (HoLs)

Continue And Break

Select Command

DolfinED
Transforming Lives Through Learning

Select Command

- The select command is a built-in construct used to easily create simple interactive shell scripts with menus.
- We define an array options containing the menu options.
- The select command displays the menu with the options listed, and the user is prompted to enter a number corresponding to their choice.
- The selected option is stored in the “var_name” variable.

Script.sh

```
#!/bin/bash
select var_name  in [LIST]
do
  [COMMANDS]
done
```

Hands-on Labs (HoLs)

Select Command

While Loop and Reading Files

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- While Loop
- Demo - While Loop
- Until Loop
- Demo – Until Loop
- Shift Command
- Demo - Shift Command
- Reading Files Line by line
- Demo - Reading Files Line by line
- Source Command
- Demo – Source Command

While Loop

DolfinED
Transforming Lives Through Learning

Explaining While Loop

- The While loop is a control structure that allows you to execute a block of code repeatedly as long as a specified condition is true.
- It is used for tasks that require repetitive execution until a certain condition becomes false.
- The while loop is particularly useful when you don't know in advance how many times the loop needs to be executed.

Script.sh

```
#!/bin/bash
while [ condition ]
do
   # Code to be executed
done
```

Explaining While Loop (cont.)

- The while keyword marks the beginning of the while loop.
- [condition] represents the condition that is tested before each iteration of the loop.
  - If the condition is true, the loop will continue executing.
  - if the condition becomes false, the loop will terminate.
- The code block enclosed between do and done will be executed repeatedly as long as the condition remains true.

Script.sh

```
#!/bin/bash
while [ condition ]
do
   # Code to be executed
done
```

Hands-on Labs (HoLs)

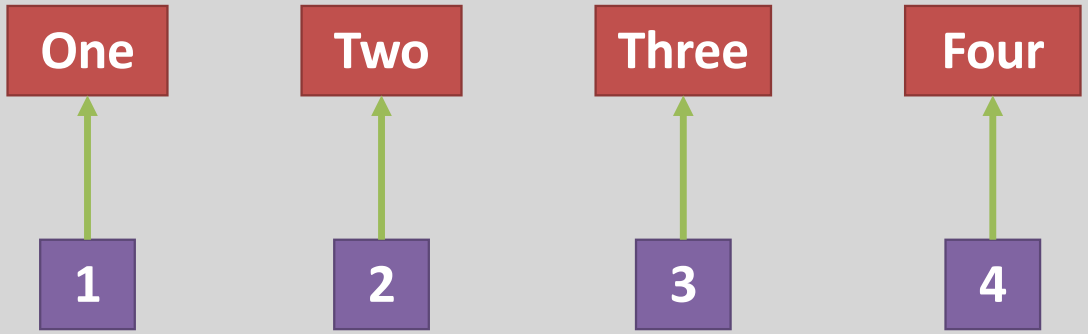
While Loop

Shift Command

DolfinED
Transforming Lives Through Learning

Shift Command

- The shift is a built-in bash command used to shift the positional parameters (arguments) passed to a script or function.
- The syntax is: shift n
  - n: The number of positions to shift. It must be a positive integer.
- The argument shifts/move the command line arguments to one position left.
  - The first argument is lost after using the shift command.



Hands-on Labs (HoLs)

Shift Command

Reading Files Line By Line

DolfinED
Transforming Lives Through Learning

Reading Files Line By Line

- To read a file line by line in Bash, you can use a while loop with the read command.
- The read command reads a single line from the standard input or a file and assigns it to a variable.
- By using a while loop, you can repeatedly read lines until the end of the file is reached.

Script.sh

```
#!/bin/bash
while IFS= read -r line
do
    # COMMAND
done < input.file
```

Reading Files Line By Line (cont.)

- The -r option of the read command prevents backslashes from being interpreted as escape characters, ensuring that the lines are read exactly as they are.
- The IFS= part clears the Internal Field Separator, which prevents leading and trailing whitespaces from being trimmed.
- The done < “input.file" part of the loop redirects the input from the file example.txt to the read command, allowing the loop to read lines from the file.

Script.sh

```
#!/bin/bash
while IFS= read -r line
do
    # COMMAND
done < input.file
```

1 : 2 : 3

Hands-on Labs (HoLs)

Reading Files Line By Line

Until Loop

DolfinED
Transforming Lives Through Learning

Explaining Until Loop

- The until loop is used to repeatedly execute a block of code as long as a certain condition remains false.
- The difference to the while loop is that a while loop executes its code block as long as the given condition is true.

Script.sh

```
#!/bin/bash
until [ condition ]
do
   # Code to be executed
done
```

Hands-on Labs (HoLs)

Until Loop

Functions

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- Explaining Functions
- Demo – Functions
- Source Command
- Demo - Source Command

Explaining Functions

DolfinED
Transforming Lives Through Learning

Explaining Functions

- A Function is a block of reusable code that performs a specific task.
- Functions allow you to break down your Bash script into smaller, modular pieces, making it easier to manage, read, and maintain.
- The commands between the curly braces {} are called the body of the function.
  - The curly braces must be separated from the body by spaces or newlines.
- When you need to execute the commands inside a function, just write the function name
- The function definition must be placed before any calls to the function.

Script.sh

```
#!/bin/bash
function_name()
{
   # code to be executed
}

function_name
```

Hands-on Labs (HoLs)

Functions in Bash

Source Command

DolfinED
Transforming Lives Through Learning

Source Command

- Using the source (or .) command inside a script allows you to include the content of another script within the current script.
- This can be useful when you want to reuse code or load environment variables or functions from another script into the current script.
- It enables you to maintain separate script files for specific purposes while leveraging shared functionality.

fun.sh

```
#!/bin/bash
function_name()
{
   # code to be executed
}
```

Script.sh

```
#!/bin/bash
source ./fun.sh

function_name
```

Hands-on Labs (HoLs)

Source Command

SED

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- Explaining SED
- Demo - SED
- Addressing and Regular Expression in SED
- Demo - Addressing and Regular Expression in SED
- SED Operators
- Demo - SED Operators

Explaining SED

DolfinED
Transforming Lives Through Learning

Explaining SED

- Sed, short for "stream editor", acts as a text editor with no interactive interface.
- The default sed command doesn’t make changes to the original file.
- The user can store the modified content into another file if needed.
- By using SED you can edit files even without opening them, which is a much quicker way to find and replace something in file.

```
sed  OPTIONS   'COMMANDS’   INPUT_FILE
```

Replacing or substituting a string

- `sed 's/<oldWord>/<newWord>/' INPUT_FILE`

s: Substitute command for search and replace.
This will replace the first word in each line.

- `sed 's/<oldWord>/<newWord>/2' INPUT_FILE`

Use the /1, /2 ..etc flags to replace the first, second ..etc word in each line.

- `sed 's/<oldWord>/<newWord>/g' INPUT_FILE`

flag /g (global replacement)
It will replace the word in all the file.

Hands-on Labs (HoLs)

SED

Addressing and Regular Expressions in SED

DolfinED
Transforming Lives Through Learning

Addressing and Regular Expression in SED

- sed ‘1 s/<oldWord>/<newWord>/g' INPUT_FILE

You can specify addresses (line numbers or patterns) to control which lines sed processes.

- sed ‘s/^<oldWord>/<newWord>/g' INPUT_FILE

“^” replaces in the line that starts with the specific word (oldWord)

- sed ‘s/<oldWord>$/<newWord>/g' INPUT_FILE

“$” replaces the lines that ENDs with the specific word

Hands-on Labs (HoLs)

Addressing and Regular Expression in SED

SED Operators

DolfinED
Transforming Lives Through Learning

SED Operators

- sed ‘s/<oldWord>/<newWord>/p' INPUT_FILE

  Duplicating the replaced line(s)

- sed -n ‘s/<oldWord>/<newWord>/p' INPUT_FILE

  Printing only the replaced lines

- sed ‘/<Word>/ d' INPUT_FILE

  Remove a complete line that has a specific word from a file

SED Operators (cont.)

- sed -e ‘<script1>’ -e ‘<script2>' INPUT_FILE

executes multiple sed commands at once

- sed -f <sed-command> INPUT_FILE

Reading sed Commands From a File

- sed -i ‘s/<oldWord>/<newWord>/' INPUT_FILE

Use –i to save the modifications to the original file

Hands-on Labs (HoLs)

SED Operators

Hands-on Labs (HoLs)

Scripts with SED

AWK

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- Explaining AWK
- Records and Fields
- Conditional Expressions
- Loops

Explaining AWK

DolfinED
Transforming Lives Through Learning

Explaining AWK

- Created by Alfred **A**ho, Peter **W**enberger, and Brian **K**ernighan, **AWK** is a powerful text processing and manipulation tool.
  - It is used for working with structured text data.
- It can be used to perform tasks such as pattern matching, filtering, sorting, and manipulating data.
- You can use regular expressions in the pattern to perform more complex matching tasks.
- AWK allows you to use built-in variables, arithmetic operations, and string manipulations in the action block to perform various text processing tasks.

The Basic Syntax of the awk command

```
awk 'BEGIN { commands } { commands } END { commands }' INPUT_FILE
```

- BEGIN: This block is optional and is executed before processing the input file.
  - It is used for initializing variables or performing any setup tasks.
  - The commands in this block are executed only once at the beginning.
- Commands: These are the actions or commands to be executed when the pattern matches the input line.
  - These commands can perform various operations on the input data, such as printing, arithmetic calculations, string manipulation, and more.
- END: This block is optional as well. It is executed after processing all the input lines.
  - It is typically used for final calculations or summary output.
  - The commands in this block are executed only once at the end.

If you want to include multiple patterns and actions, you can separate them using semicolons ;

Hands-on Labs (HoLs)

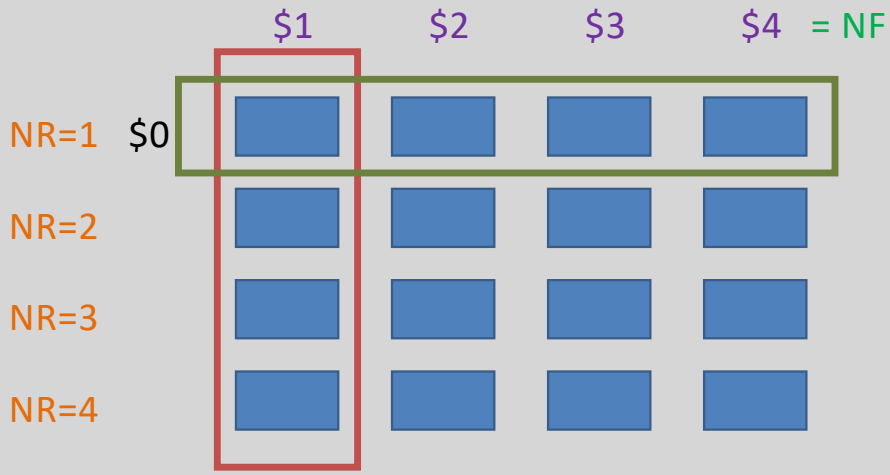
AWK

AWK Variables

DolfinED
Transforming Lives Through Learning

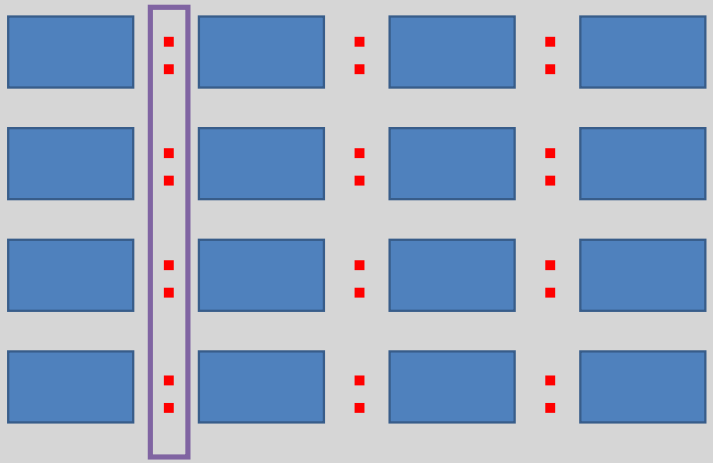
AWK Variables

Variable	Description
NR	**Current record number.** It represents the total number of records (lines) processed so far.
NF	**Number of fields in the current record.** It represents the total number of fields separated by the field separator.
$0	**Represents the whole line**
$1, $2, ….	**Represent the positions in a line.** If the line has 4 words, it will be stored in $1, $2, $3 and $4 positions respectively



AWK Variables

Variable	Description
FS	**Field separator.** It specifies the character or regular expression that separates fields in the input data. By default, it is any whitespace (spaces or tabs).
OFS	**Output field separator.** It specifies the character used to separate fields in the output. By default, it is a space.
RS	**Record separator.** It determines how awk identifies the end of a record (line) in the input data. By default, it is a newline.
ORS	**Output record separator.** It determines how awk separates records in the output data. By default, it is a newline.



Hands-on Labs (HoLs)

AWK Variables – Part 1

Hands-on Labs (HoLs)

AWK Variables – Part 2

Hands-on Labs (HoLs)

Operations in AWK

Hands-on Labs (HoLs)

Operations in AWK – Part 2

If Statements in AWK

DolfinED
Transforming Lives Through Learning

If Statement in AWK

- The awk command is enclosed in single quotes (') to prevent the shell from interpreting the script's content.
- Inside the awk script, you can define the pattern to match lines or records from the input.
- Within the curly braces {}, you can use the if statement to check a condition.
- The action inside the if block will be executed only when the condition is true.
- Optionally, you can have an else block with its own action to be executed when the condition is false.

Script.sh

```
#!/bin/bash
awk '{
  if (condition) {
    # Action to be executed
  }
  else {
    # Action to be executed
  }
}' input_file
```

Hands-on Labs (HoLs)

If Statement in AWK

For Loop in AWK

DolfinED
Transforming Lives Through Learning

For Loop in AWK

- The for loop in awk works similarly to for loops in other programming languages, and it allows you to perform repetitive actions on data elements within a record.
- variable: It is a temporary variable that takes on the value of each element in the collection during each iteration of the loop.
- collection: It can be an array, a list of fields in the current record (such as $1, $2, etc.), or other data structures.

Script.sh

```
#!/bin/bash
awk ' {
   for (variable in collection)
   {
      # Action to be performed for each
        element in the collection
   }
} ' input_file
```

Hands-on Labs (HoLs)

For Loop in AWK

Scheduling The Scripts

DolfinED
Transforming Lives Through Learning

Section Outline

In this section, we will learn:

- "at" Command
- Demo - "at" Command
- Crontab
- Demo - Crontab

"at" Command

DolfinED
Transforming Lives Through Learning

"at" Command

The at command is used to schedule one-time tasks or script executions to occur at a specific time in the future.

at 15:30	Schedule at 3:30 PM today
at 15:30 –f <script-name>	Schedule the specific script at 3:30 PM today
atq	List the Schedule jobs
atrm 1	Remove a Scheduled Job followed by the job number

Hands-on Labs (HoLs)

"at" Command

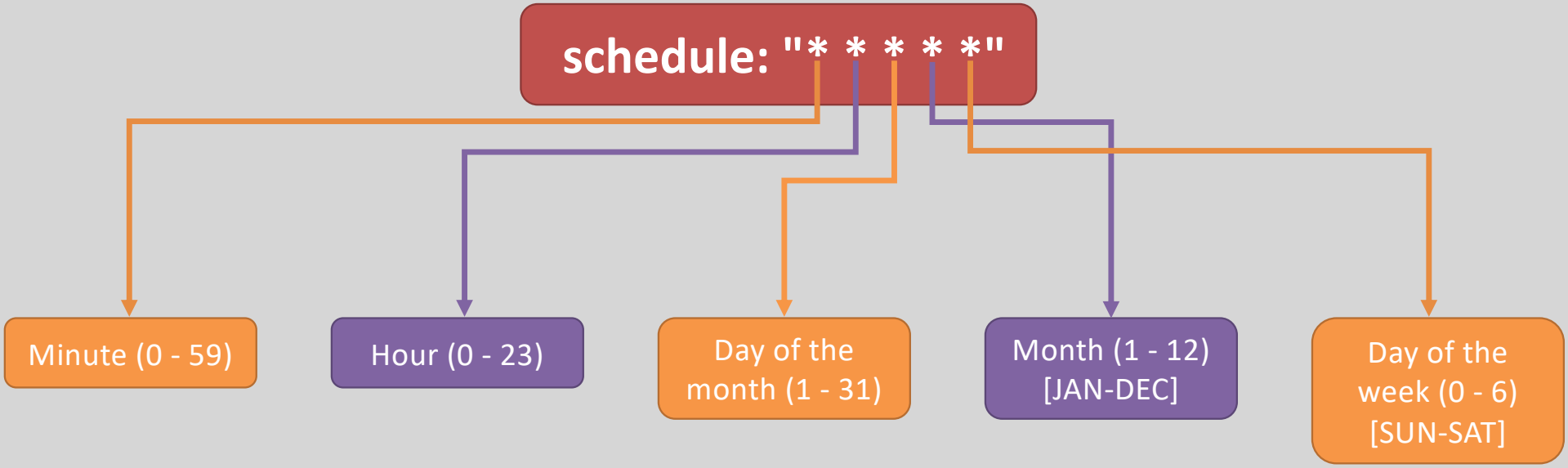
“Crontab” Command

DolfinED
Transforming Lives Through Learning

“Crontab” Command

- Cron is a powerful time-based job scheduler in Unix-like operating systems, including Linux and macOS.
  - It allows you to schedule scripts or commands to run at specific times or on specific intervals.
- To schedule tasks for your user,
  - Run the command “crontab –e”
- In the crontab file, each line represents a scheduled task.
  - The format of a cron entry is as follows
    - “ * * * * * command_to_be_executed “

Cron Schedule Syntax



Calculate Schedule

https://crontab.guru/

Hands-on Labs (HoLs)

“Crontab” Command

Capstone Project

DolfinED
Transforming Lives Through Learning

Capstone Project

Using A Bash Script and Scheduling to Automate Virtual Machine Data Backup & Replication To An Amazon S3 Bucket

DolfinED
Transforming Lives Through Learning

Automating Virtual Machine Backups & Replication To An Amazon S3 Bucket

Create a bash script that runs on a schedule to compress and backup critical data on a virtual machine (EC2 Instance) in AWS.

- Automate the backup replication to an Amazon S3 Bucket.

- Key Features:
  - Backup Scheduling: Allow users to schedule backups at specific times or intervals (e.g., daily, weekly, or monthly)
  - Backup Compression: Compress the backup files to save storage space using the tar command with gzip (tar -czf) or other compression methods.
  - Logging: Create log files to record backup operations, including start time, end time, and any errors encountered during the backup.
  - AWS Integration: Seamlessly integrate with the AWS Command Line Interface (CLI) for secure and efficient uploads to Amazon S3.

Automating Virtual Machine Backups & Replication To An Amazon S3 Bucket (cont.)

Capstone Project

User Account Management Script

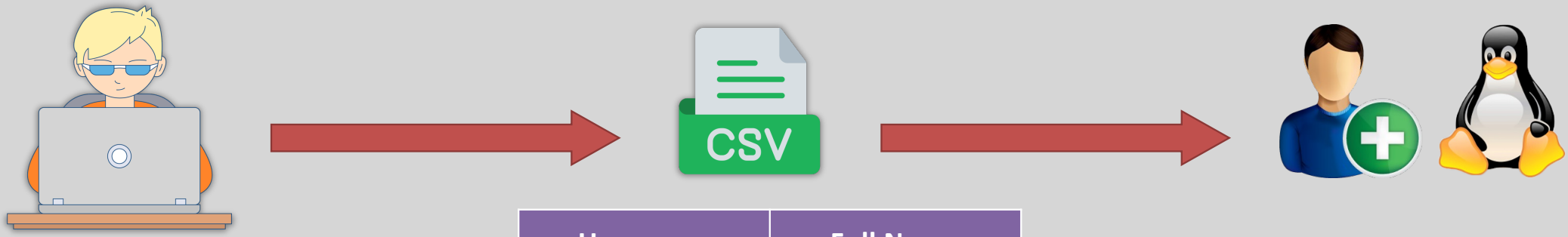
User Account Management Script

Create a Bash script to automate the creation of user accounts, generate random initial passwords, and enforce password change on first login.

- The script reads user data from a CSV file, allowing for the bulk provisioning of user accounts with enhanced security.

- Key Features:
  - Create A CSV file: Create a CSV file with a usernames and full-names.
  - CSV Data Input: Read user data from a CSV file for batch user account creation.
  - User Account Creation: Create user accounts with specified usernames and full names.
  - Random Password Generation: Generate strong, random passwords for each user.
  - Root Privilege Check: Ensure that the script is run with root privileges for user management.

User Account Management Script



Username	Full Name
aws	Amazon Web Services
K8s	Kubernetes