BASH SCRIPTING
OR
SHELL PROGRAMMING

LOGICOPS LAB BY RAVISH RAWAT

ROADMAP

- Need of Shell scripting
- What is Shell / What is Bash?
- Use of Variables and Comments
- Read User Input
- Pass Arguments to Bash script
- If Statement(If then, if then else, if elif else)
- File test operators
- How to append output to the end of file
- Logical "AND" "OR" operator

ROADMAP (CONTD)

- Arithmetic operations
- The Case statement
- While loop
- UNTIL loop
- For loop
- Select loop
- Break and Continue
- Functions
- Interview questions & programs hands-on

PRE-REQUISITES

- Oracle Virtual Box
- CentOS vdi (Virtual Desktop Infrastructure)
- Internet
- How to install and configure the mentioned softwares

GUEST ADDITIONS

- Guest Additions

  - The VirtualBox Guest Additions consist of device drivers and system applications that optimize the operating system for better performance and usability. One of the usability features required in this guide is automated logons, which is why you need to install the Guest Additions in the virtual machine.

- How to add Guest Additions? (Installations and Commands)

INITIAL SETUP

- Initial setup to make the environment ready
- Writing and Running our first Bash Script
- Various ways of executing a shell file in a Linux environment
- How to get icons visible in CentOS on Desktop? GNome Tweaks.
- How to install git in Linux and How to clone a repository in Linux?
- How to install Visual Studio Code in Linux? How to run a script using Visual Studio Code

INTRODUCTION

- What are Scripts?
- Components of a script.
- Variable usage in the scripts.
- How to accept input from a user?
- How to make decisions?
- Performing tests and a lot more.

SCRIPTS

- Series of commands to achieve something
- These commands are executed by an interpreter (Here shell acts as an interpreter)
- Any command in a format can be written into a script
- Why do we need it? – To automate mundane tasks and save some time.
- Example

```
#!/bin/bash
echo “Hello, World”
```

(Need of #! Before the script, the script runs using your shell. It might or might not run, as different shells have a bit of varying syntaxes)

VARIABLES

- Storage location having a name or a Name-Value pair
- Case sensitive
- No spaces before or after the = sign
- Uppercase
- Syntax:
  - VAR_NAME=“Any Value”
  - Eg –

```
#!/bin/bash
SHELL_NAME=“bash”
echo “I am $SHELL_NAME” ->  I am bash
Or echo “I am ${SHELL_NAME}ing -> I am bashing
```

(If you don’t give a curly brace, the shell will treat it as an additional text)

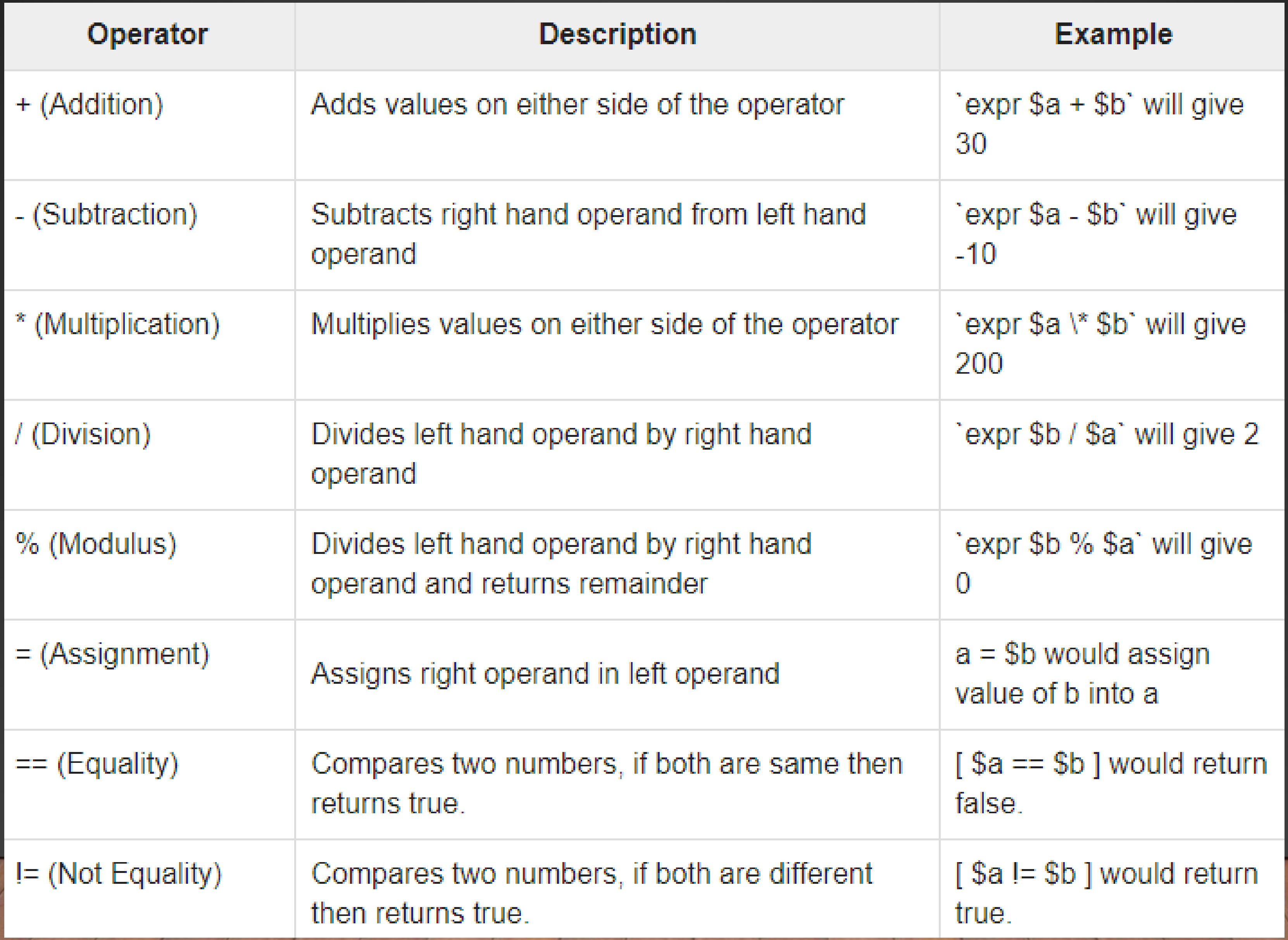
VARIABLES (CONTD)

- Things to remember while declaring a Variable:
- Not Valid
  - 1One="Ravish"
  - one@two="LogicOpsLab"
  - ONE-TWO="Rawat"
- Valid
  - ONETWO="RAVISH"
  - ONE_TWO="LogicOpsLab"
  - oneTwo="Rawat"

OPERATORS

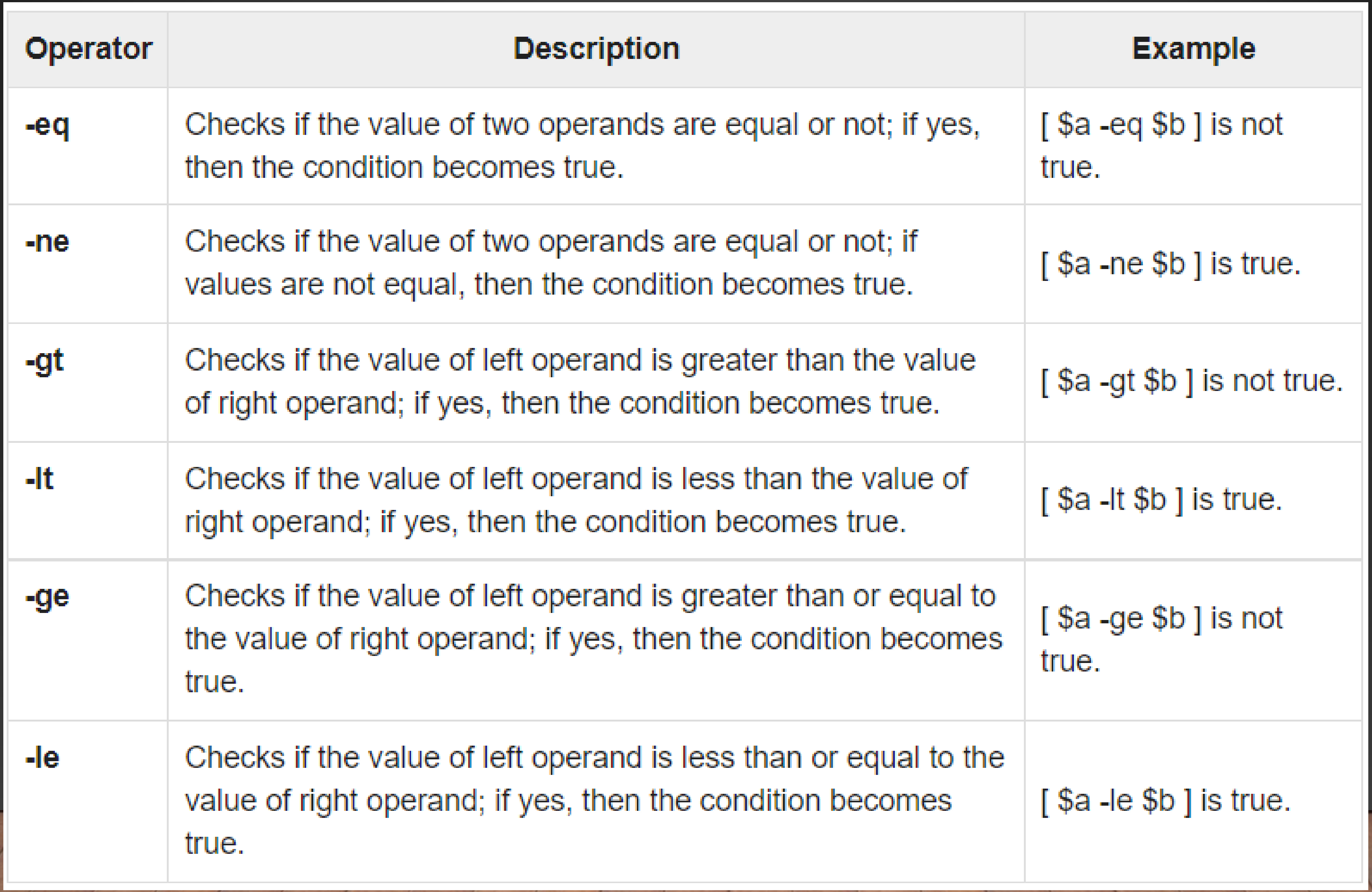
- We have 5 types of operators
  - Arithmetic Operators
  - Relational Operators
  - Boolean/Logical Operators
  - Bitwise Operators
  - File Test Operators

ARITHMETIC OPERATORS



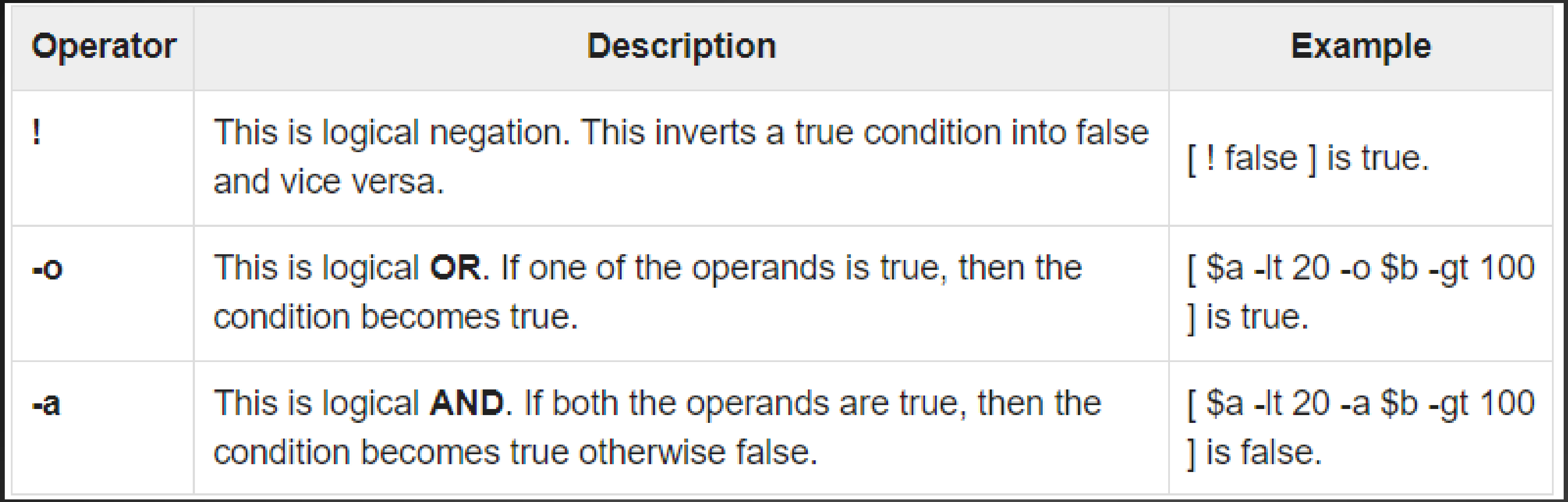
Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

RELATIONAL OPERATORS



Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

BOOLEAN/LOGICAL OPERATORS



Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

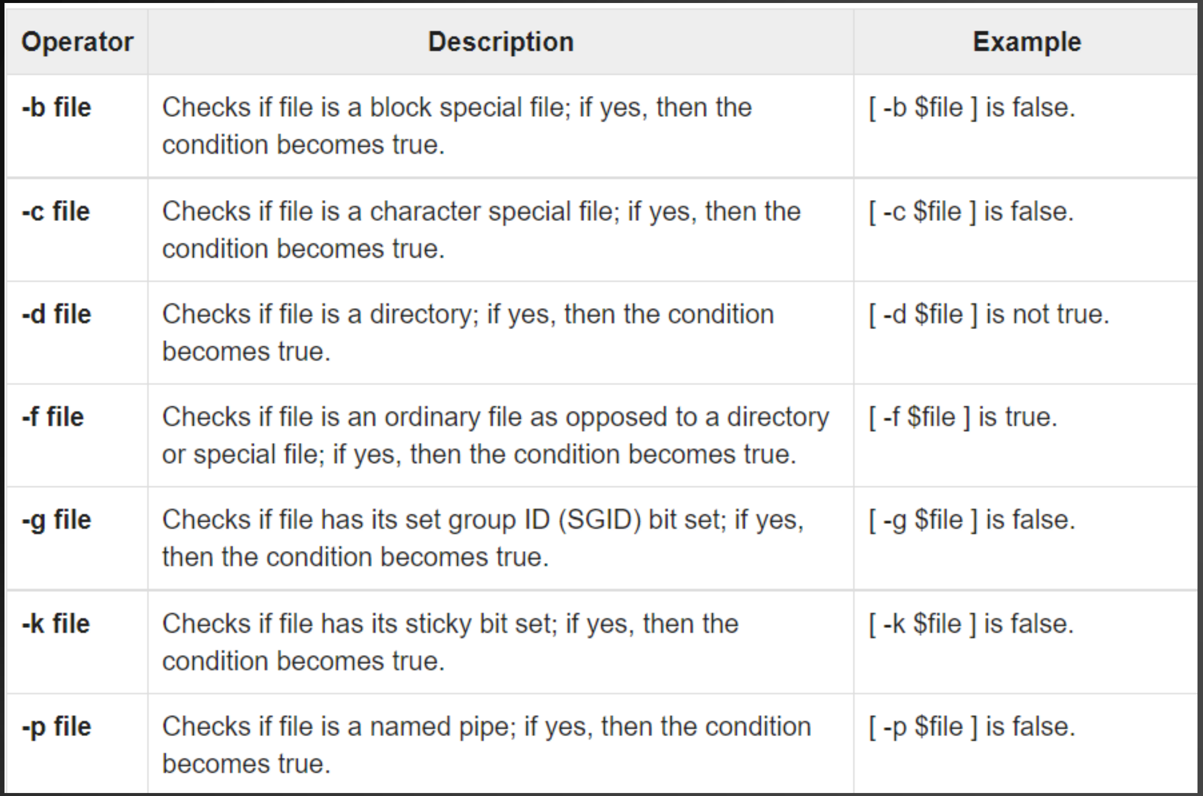
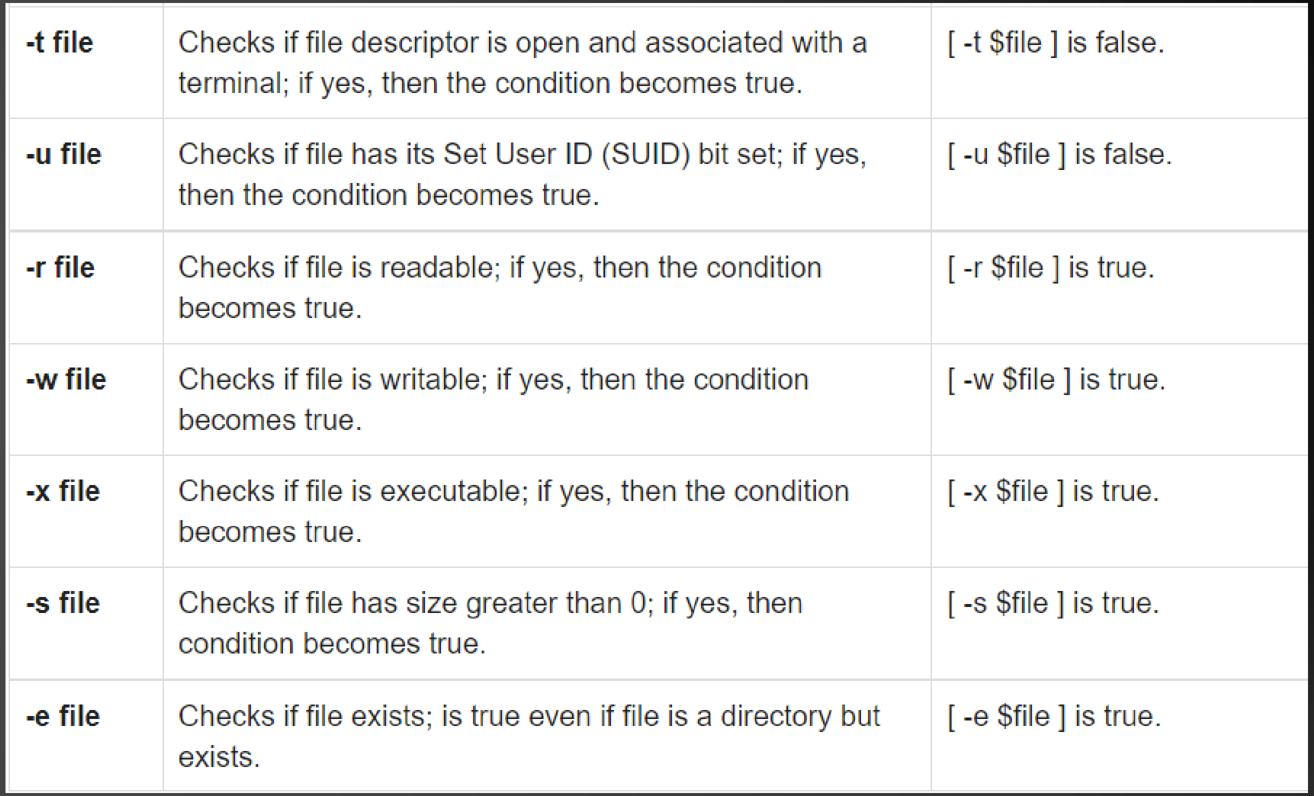
Logical OR (||)

Logical AND (&&)

BITWISE OPERATORS

- **Bitwise And (&):** Bitwise & operator performs binary AND operation bit by bit on the operands.
- **Bitwise OR (|):** Bitwise | operator performs binary OR operation bit by bit on the operands.
- **Bitwise XOR (^):** Bitwise ^ operator performs binary XOR operation bit by bit on the operands.
- **Bitwise complement (~):** Bitwise ~ operator performs binary NOT operation bit by bit on the operand.
- **Left Shift (<<):** This operator shifts the bits of the left operand to left by number of times specified by right operand.
- **Right Shift (>>):** This operator shifts the bits of the left operand to right by number of times specified by right operand.

FILE TEST OPERATORS

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

MAKING DECISIONS / CONDITIONAL STATEMENTS

- There are total of 5 Conditional Statements in Bash Scripting/Programming
  - if statement
  - if-else statement
  - if..elif..else..fi statement (Else If ladder)
  - if..then..else..if..then..fi..fi..(Nested if)
  - switch statement

TESTS

- Test is used by virtually every shell script written. It may not seem that way, because test is not often called directly.
- Syntax:
  - [test-condition]
  - Test condition can be anything; a number comparison, a file exists or not, etc
- Example
  - [-e /etc/passwd]
  - This will check if the /etc/passwd exists or not. Returns TRUE if it does, FALSE if there is no file.
- Test is most often invoked via the if-else and the while loops-statements

MAKING DECISIONS

- If Statement
- Syntax
    - if [true-condition]

      then

      do this

      do that

      fi
- Here, fi is just if spelled backwards

MAKING DECISIONS (CONTD)

- If-else Statement
- Syntax

  -
    ```
    if [ true-condition ]
      then
                  do this
     else
                  do that
     fi
    ```

MAKING DECISIONS (CONTD)

- Else if ladder
- Syntax

  -
    ```
    if [ expression1 ]
        then
        statement1
        statement2
        .
        elif [ expression2 ]
        then
          statement3
          statement4
          .
        else
          statement5
        fi
    ```

MAKING DECISIONS (CONTD)

- Nested if
- if [expression1]

```
then
   statement1
   statement2
   .
else
   if [ expression2 ]
   then
      statement3
      .
   fi
fi
```

MAKING DECISIONS (CONTD)

- switch case
- Syntax

```
case  in
  Pattern 1) Statement 1;;
  Pattern n) Statement n;;
esac
```

RETURN CODE & EXIT STATUS

- What is exit status or a return code?
- Checking exit status
- What to do on the basis of the status?
- Can we use exit status in scripts?
- Demo

RETURN CODE & EXIT STATUS

- Every command returns an exit status
- It ranges from 0 (which is success) to 255
- Anything apart from 0 is an error condition
- We can use commands like info and man to understand the meaning of an exit status
- Explicitly define Return Codes – exit 0 to exit 255
- You can use exit anywhere in the shell script.

FUNCTIONS

- What are Functions?
- Why Functions?
- How to declare and use them?
- What is a variable scope?
- How to call a Function?
- Parameters in a Function
- Exit status and Return codes
- Demo

FUNCTIONS

- A function is a block of code that is reusable, performs certain operations, and gives you a result.
- Functions are popular because:
  - DRY – Don't Repeat Yourself.
  - Help to reuse a piece of code, thus reduce script length
  - Improves the readability
  - Modular approach
  - Easy maintenance as we have single place to edit and troubleshoot
- Whenever you are repeating a code, write a function
- Functions can call other functions.

FUNCTIONS

- **Creation / Declaration**
  - Function function-name( )

```
{
        # Your code here
}
```

- Or
  - Function-name( )

```
{
        # Your code here
}
```

- Calling a Function
  - Function-name

    (That is all, no brackets/paranthesis like other languages)

WILDCARDS

- What are Wildcards?
- Where they can be used?
- How can we use them?
- Types of Wildcards
- Ranges in Wildcards
- Demo

WILDCARDS

- Wildcards (also referred to as meta characters, glob characters) are symbols or special characters that represent other characters
- You can club it with some command for usage, like ls, rm, cp
- Types :
- Asterisk (*) – matches one or more occurrences of any character, including no character.
- Question mark (?) – represents or matches exactly one character.
- Bracketed characters / Character class ([ ]) – matches any occurrence of character enclosed in the square brackets. You can use different types of characters. Like, numbers, letters, other special characters, etc.
- [!] – Matches any of the characters NOT included between the brackets

WILDCARDS

- Ranges – You can use 2 characters separated by a hyphen to create a range
- [a-d]* - Will match all the files that starts with a, b, c, or d

- Named character classes – They are predefined
  - [[:upper:]] - matches any uppercase letters
  - [[:lower:]] – matches any lowercase letters
  - [[:digit:]] - matches numbers from 0-9
  - [[:space:]] - matches whitespace like spaces and tabs
  - [[:alpha:]] - matches alphabetical letters, both upper and lower case
  - [[:alnum:]] - matches alphanumeric characters, both upper and lower case, decimal also

LOOPS

- What is a Loop?
- Types of Looping statement or Loops
- Statements – break and continue
- Usage of loops with break and continue
- Demo with examples

LOOPS

- Loop –
  - A loop is a sequence of instructions that is repeated continuously until a certain condition is matched or reached.
  - If the condition has not matched, the next instruction in the sequence is an instruction to return to the first instruction in the sequence and repeat the sequence.
  - If the condition has been met, the next instruction will fall through to the next sequential instruction or it will branch outside the loop.
- Infinite loop – A loop that does not have an exit condition, which is why it keep on repeating itself.
- Types –
  - While loop
  - For loop
  - Until statement

SYNTAX

- While –
  - while

    do

    Your code here

    done
- For –
  - for var in word1 word2 till wordn

    do

    Your code here

    done
- Until –
  - until

    do

    Your code that needs to be executed until command is true

    done

LOGGING

- What is Logging?
- Why do we need logging?
- How to generate logs?
- syslog
- Demo

LOGGING

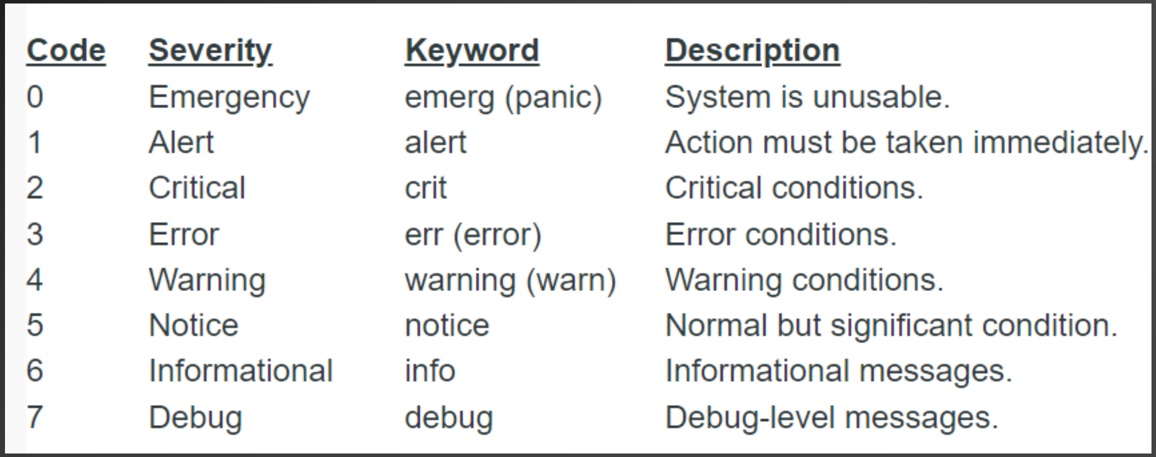
- Helpful when there are no debugging tools
- Logs help localize a problem and narrow it down
- Logs help to check if everything is working as expected
- Helps figure out if there is a security concern like malware, virus, trojan, unexpected intrusion, etc.
- If you run your scripts unattended; Logging plays an important role
- Syslog-ng is a very famous tool that helps security experts, system admins, and DevOps managers in centralizing all of the log messages coming from servers, network devices, applications, printers, etc
- Reduces downtime

LOGGING (SYSLOG)

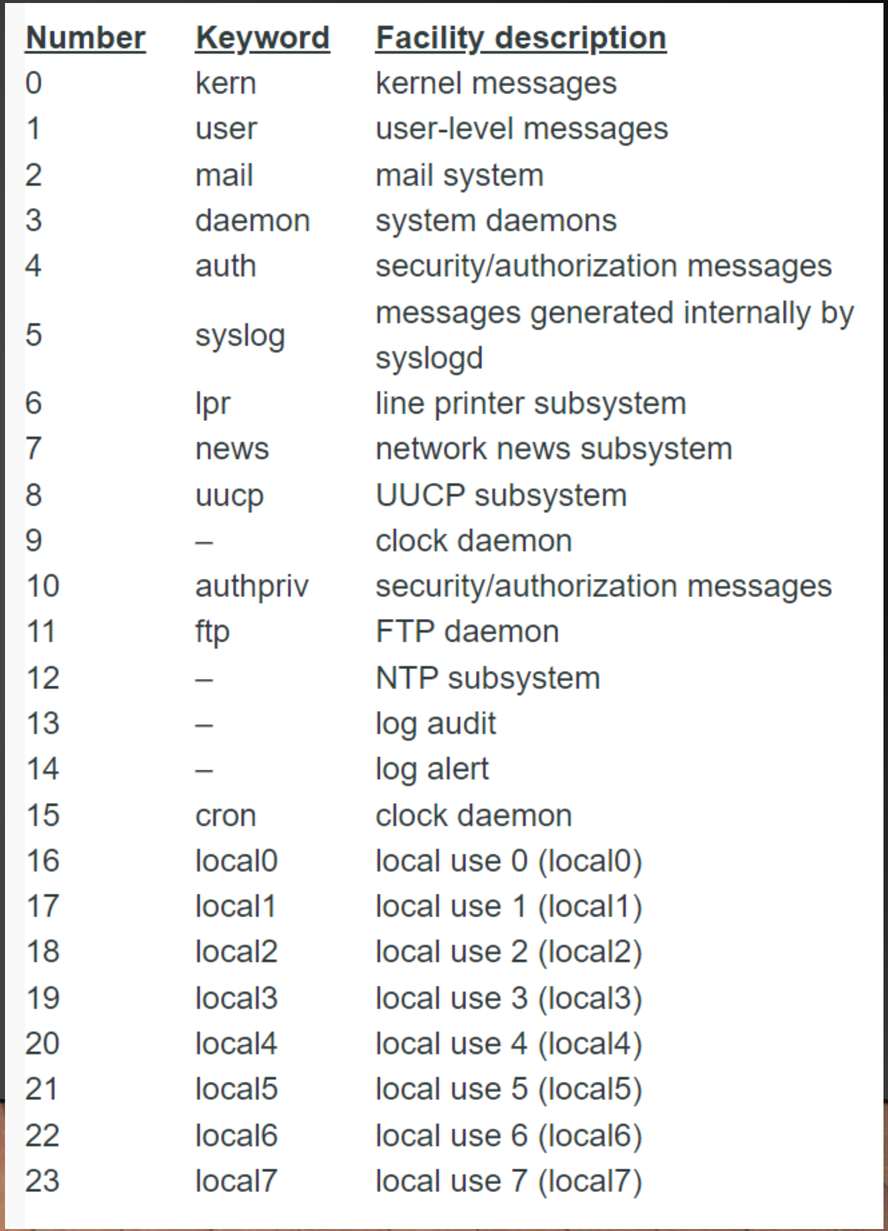
- Syslog is a protocol for logging and tracking system messages in linux
- Application use syslog to export all their error, status messages, to the file in the /var/log directory (which is configurable)
- Components
  - Listener - Gathers and processes syslog data
  - Database – To store the data
  - Management and filtering software

LOGGING (SYSLOG)

- Logsys uses Facilities and Severities to categorize messages
- Note that, when you specify a priority, you are actually specifying everything at that priority or higher. For example, mail.info would mean all messages coming from the mail facility with the info, notice, warning, err, crit, alert or emerg priority.



Code	Severity	Keyword	Description
0	Emergency	emerg (panic)	System is unusable.
1	Alert	alert	Action must be taken immediately.
2	Critical	crit	Critical conditions.
3	Error	err (error)	Error conditions.
4	Warning	warning (warn)	Warning conditions.
5	Notice	notice	Normal but significant condition.
6	Informational	info	Informational messages.
7	Debug	debug	Debug-level messages.



Number	Keyword	Facility description
0	kern	kernel messages
1	user	user-level messages
2	mail	mail system
3	daemon	system daemons
4	auth	security/authorization messages
5	syslog	messages generated internally by syslogd
6	lpr	line printer subsystem
7	news	network news subsystem
8	uucp	UUCP subsystem
9	–	clock daemon
10	authpriv	security/authorization messages
11	ftp	FTP daemon
12	–	NTP subsystem
13	–	log audit
14	–	log alert
15	cron	clock daemon
16	local0	local use 0 (local0)
17	local1	local use 1 (local1)
18	local2	local use 2 (local2)
19	local3	local use 3 (local3)
20	local4	local use 4 (local4)
21	local5	local use 5 (local5)
22	local6	local use 6 (local6)
23	local7	local use 7 (local7)

CASE STATEMENTS

- One way to work on a multiway branch is that you can use multiple if...elif statements. But, sometime, it is not the best solution, especially when all of the branches depend on the value of a single variable.
- For that we have **case...esac** statement. This handles exactly this type of situation. While comparing, **case...esac** does so more efficiently than repeated if...elif statements.
- Syntax :

```
case word in
  pattern1)
    Statement(s) to be executed if pattern1 matches
    ;;
  pattern2)
    Statement(s) to be executed if pattern2 matches
   ;;
*)
   Default condition to be executed
   ;;
esac
```

CASE STATEMENTS

- Each case statement starts with the case keyword, followed by the case expression and the in keyword. The statement ends with the esac keyword
- You can use multiple patterns separated by the | operator, Using) operator we terminate a pattern list
- A pattern can have special characters
- A pattern and its associated commands are known as a clause which must be terminated with ;;
- The code block of first pattern that matches the expression will be executed
- Wildcard asterisk symbol (*) should be the final pattern to define the default case. This pattern will always match

DEBUGGING

- What is Debugging?
- How to Debug?
- Editing in Terminal
- Debugging options
- DEBUG method

DEBUGGING

- Debugging - In computer programming and software development, debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems.
- Editing in Terminal – VI editor
- Debugging options
  - #! /bin/bash –x **OR** #! /bin/bash –v **OR** #! /bin/bash –xv
- Separately
  - -x (xtrace or execution trace)
  - -v (verbose)
  - -u (unset)
  - -n (noexec)
- DEBUG function : _DEBUG="on"

SED – STREAM EDITOR

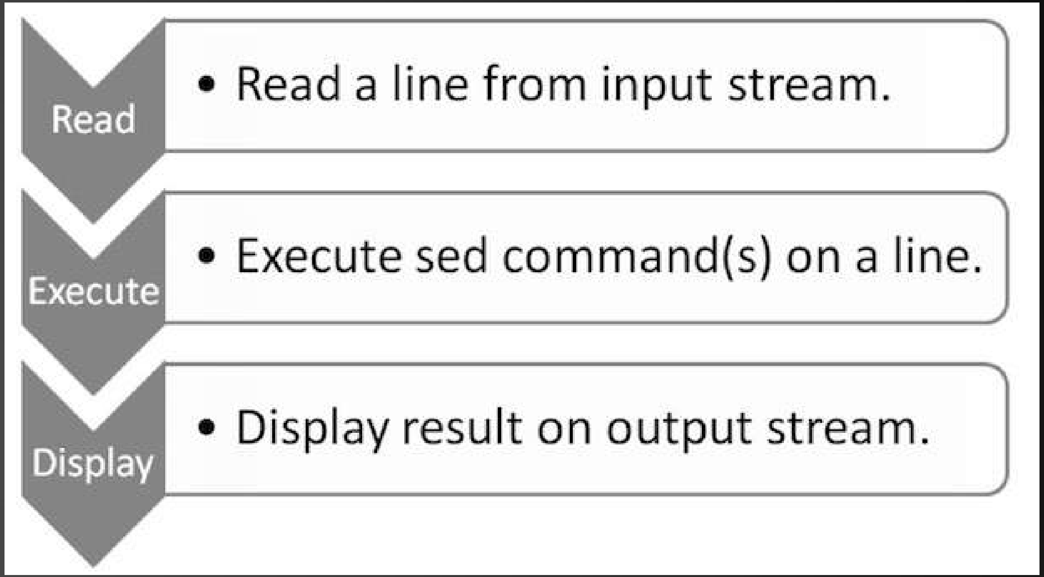
- What is SED?
- Usage of SED
- Syntax
- Workflow
- Standard Options
- Demo

SED – STREAM EDITOR - INTRODUCTION

- It performs editing operations on text coming from a file or standard input.
- sed edits line-by-line and in a non-interactive way.
- This means that you make all of the editing decisions as you are calling the command, and sed executes the directions automatically.
- This tutorial will cover some basic operations and introduce you to the syntax required to operate this editor.

SED – STREAM EDITOR - USAGE

- SED performs text transformations on streams
- Substituting text for some other text
- Line removal
- Appending text before and after the given lines
- Non-interactive editing of text files
- Syntax
  - ‘s / Search Pattern / Replacement /’ filename
  - ‘s / Search Pattern / Replacement /i’ filename – I or i is ignore case
  - ‘s / Search Pattern / Replacement /g’ filename – g is global, more than one occurrences


Read
• Read a line from input stream.
Execute
• Execute sed command(s) on a line.
Display
• Display result on output stream.