DevOps Cloud Camp
keep on practicing

Linux

Shell Scripting

Prepared BY
Aravind Kumar G.K
Senior DevOps Engineer

Agenda

- What is Shell?
- Types of Shell
- Extended Shell Scripts
- Variables
- Arrays
- Basic Operators
- Loops

What is Shell?

- Shell is an environment in which we can run our commands, programs, and shell scripts.
- There are different flavors of a shell, just as there are different flavors of operating systems.
- Each flavor of shell has its own set of recognized commands and functions.
- The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.
- Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input.
- A word is an unbroken set of characters. Spaces and tabs separate words.

DevOps Cloud Camp
keep on practicing

Shell Types

- In Unix, there are two major types of shells −
- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

Shell Types

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

Variables

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
- Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

Variable Names

- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- By convention, Unix shell variables will have their names in UPPERCASE.

Valid Variable Names	Invalid Variable Names
_ALI	2_VAR
TOKEN_A	-VARIABLE
VAR_1	VAR1-VAR2
VAR_2	VAR_A!

DevOps Cloud Camp
keep on practicing

Variable Types

- **Local Variables** – A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** – An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** – A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Defining Variables

Variables are defined as follows −

variable_name=variable_value

For example − > NAME=“Aravind"

Accessing Values

- To access the value stored in a variable, prefix its name with the dollar sign ($)

- For example, the following script will access the value of defined variable NAME and print it on STDOUT

```
#!/bin/sh
NAME=" Aravind"
echo $NAME
```

The above script will produce the following value

```
Aravind
```

DevOps Cloud Camp
keep on practicing

Read-only Variables

- Shell provides a way to mark variables as read-only by using the read-only command.
- After a variable is marked read-only, its value cannot be changed.

```
#!/bin/sh

NAME="Aravind"
readonly NAME
NAME="Aravind"
```

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

- Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks.
- Once you unset a variable, you cannot access the stored value in the variable.
- Following is the syntax to unset a defined variable using the unset command

```
unset variable_name
```

```
#!/bin/sh
NAME="Zara Ali"
unset NAME
echo $NAME
```

This example does not print anything.
You cannot use the unset command to unset variables that are marked read-only

DevOps Cloud Camp
keep on practicing

Special Variables

- Variables and descriptions

Usage of $S, $!, S* and $@
$$ →gives the process id of the currently executing process whereas
$! →Shows the process id of the process that recently went into the background.
$* → gives all passed arguments to the script as a single string
$@ → gives all passed arguments to the script as delimited list. Delimiter $IFS
What is usage of $1 and $2
First argument: $1,
Second argument : $2
Usage of $#, $0, S?
$# → To calculate number of passed arguments
$0 →To get script name inside a script (file name)
$? → To check if previous command run successful

DevOps Cloud Camp
keep on practicing

Arrays

- Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time.

- Arrays provide a method of grouping a set of variables.

- Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

DevOps Cloud Camp
keep on practicing

Defining Array Values

- Suppose you are trying to represent the names of various students as a set of variables.
- Each of the individual variables is a scalar variable as follows

```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy
```

We can use a single array to store all the above mentioned names

```
array_name[index]=value
```

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

Accessing Array Values

- After you have set any array variable, you access it as follows

```
${array_name[index]}
```

```
#!/bin/sh
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

Output

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

```
#!/bin/sh
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

You can access all the items in an array in following ways

Output

```
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

Shell Basic Operators

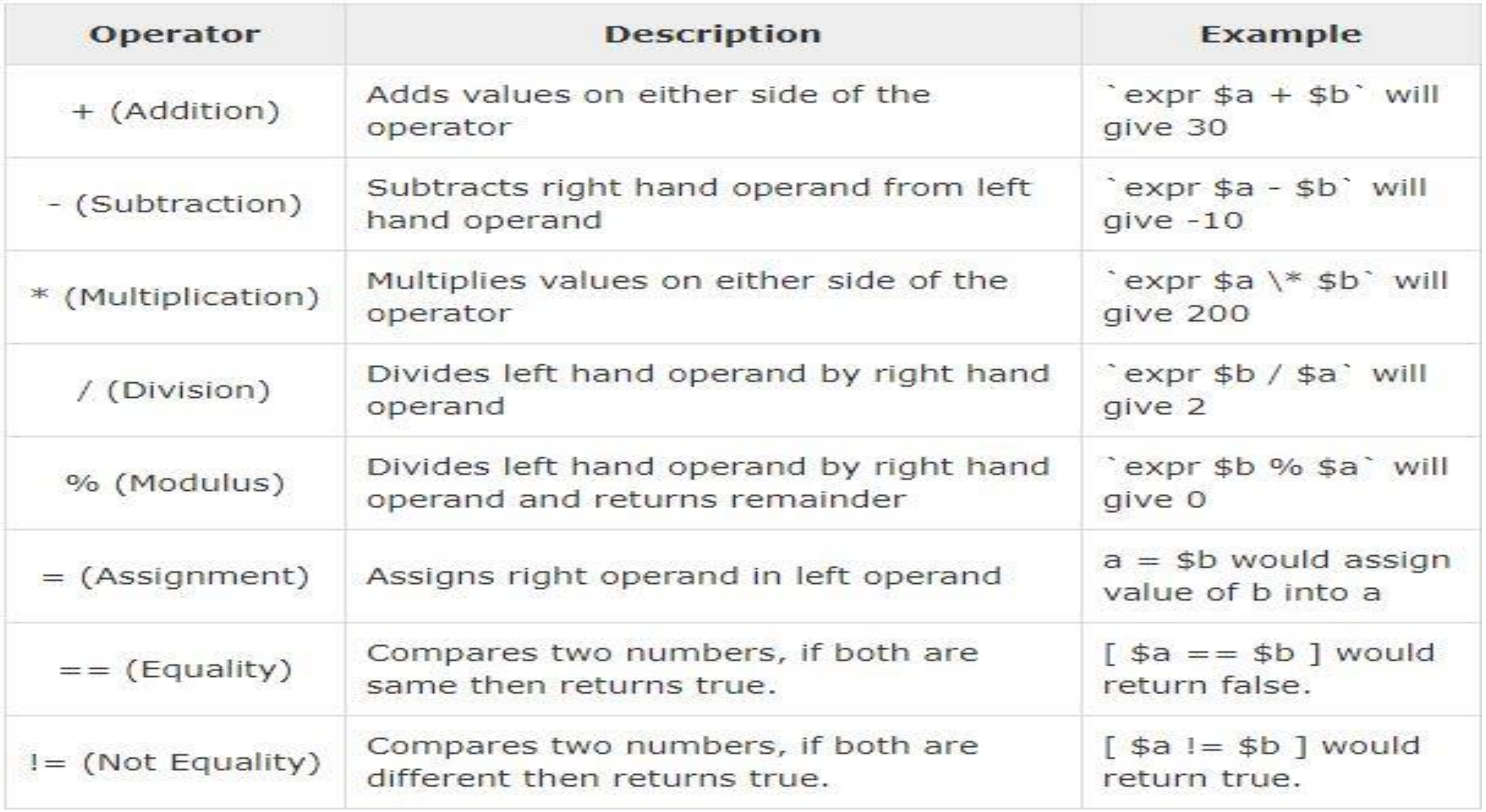
- There are various operators supported by each shell.
- We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

DevOps Cloud Camp
keep on practicing

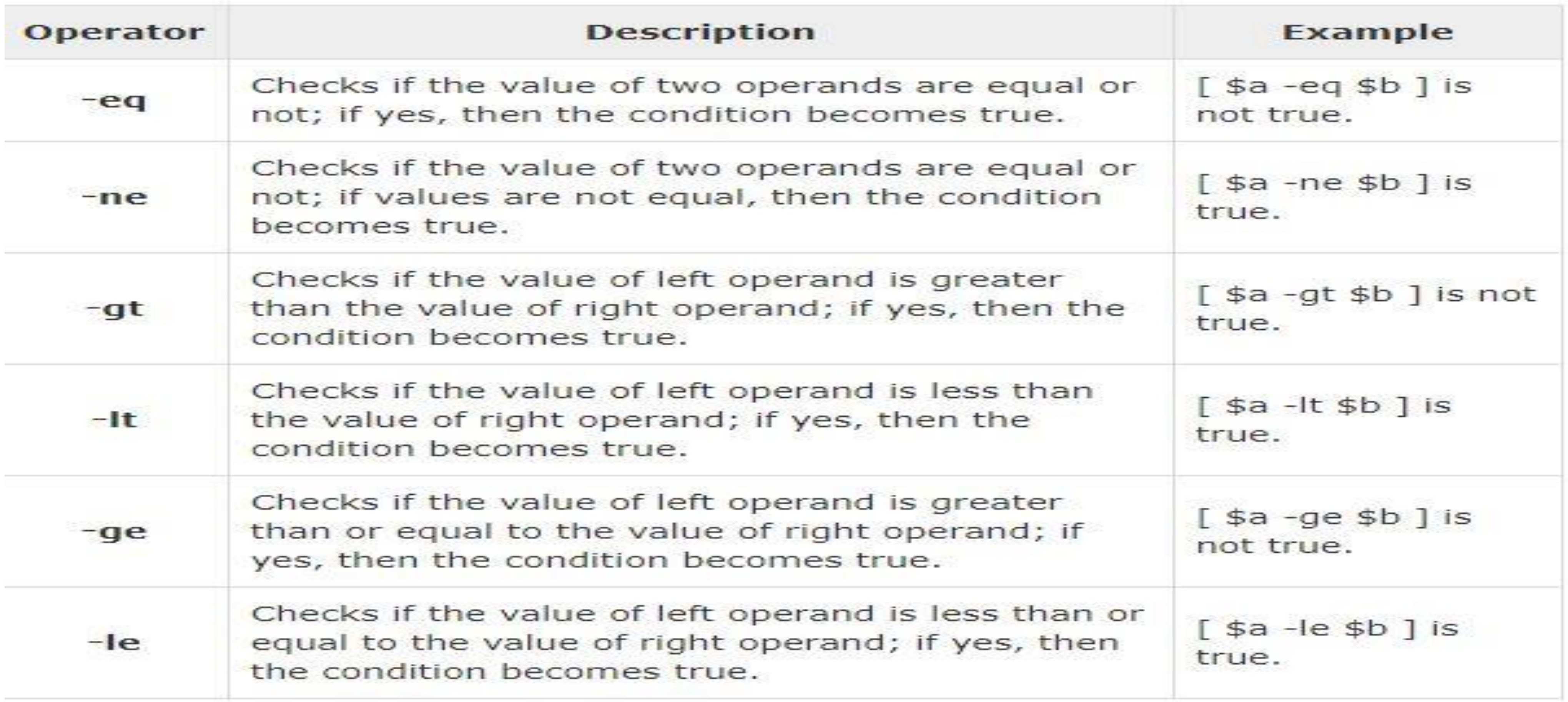
Arithmetic Operators



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.

DevOps Cloud Camp
keep on practicing

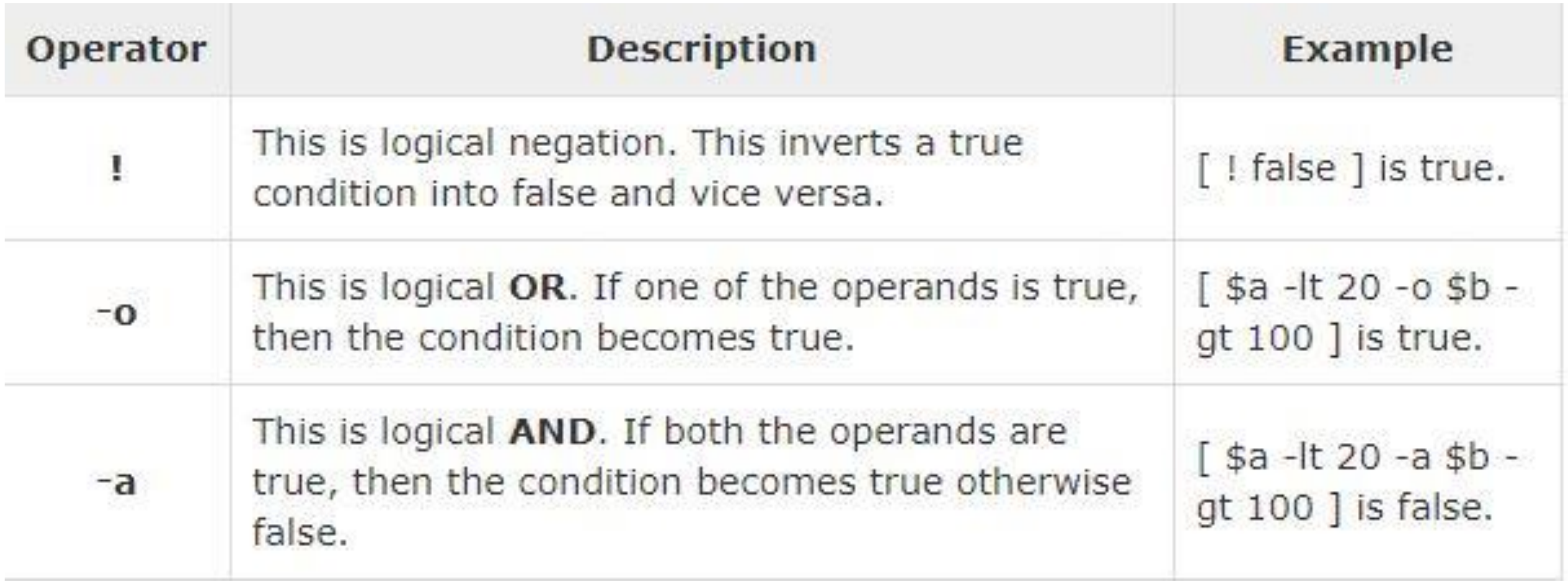
Relational Operators



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.

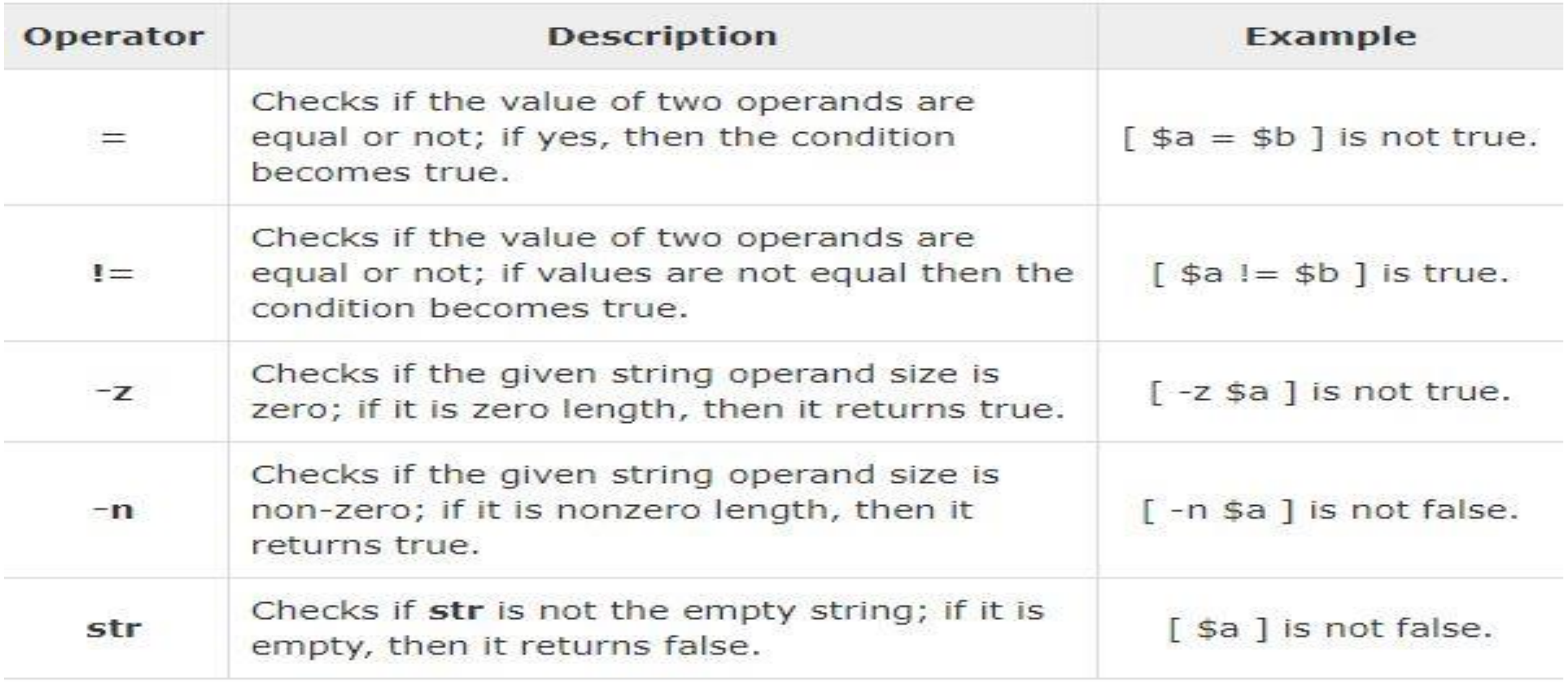
DevOps Cloud Camp
keep on practicing

Boolean 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.

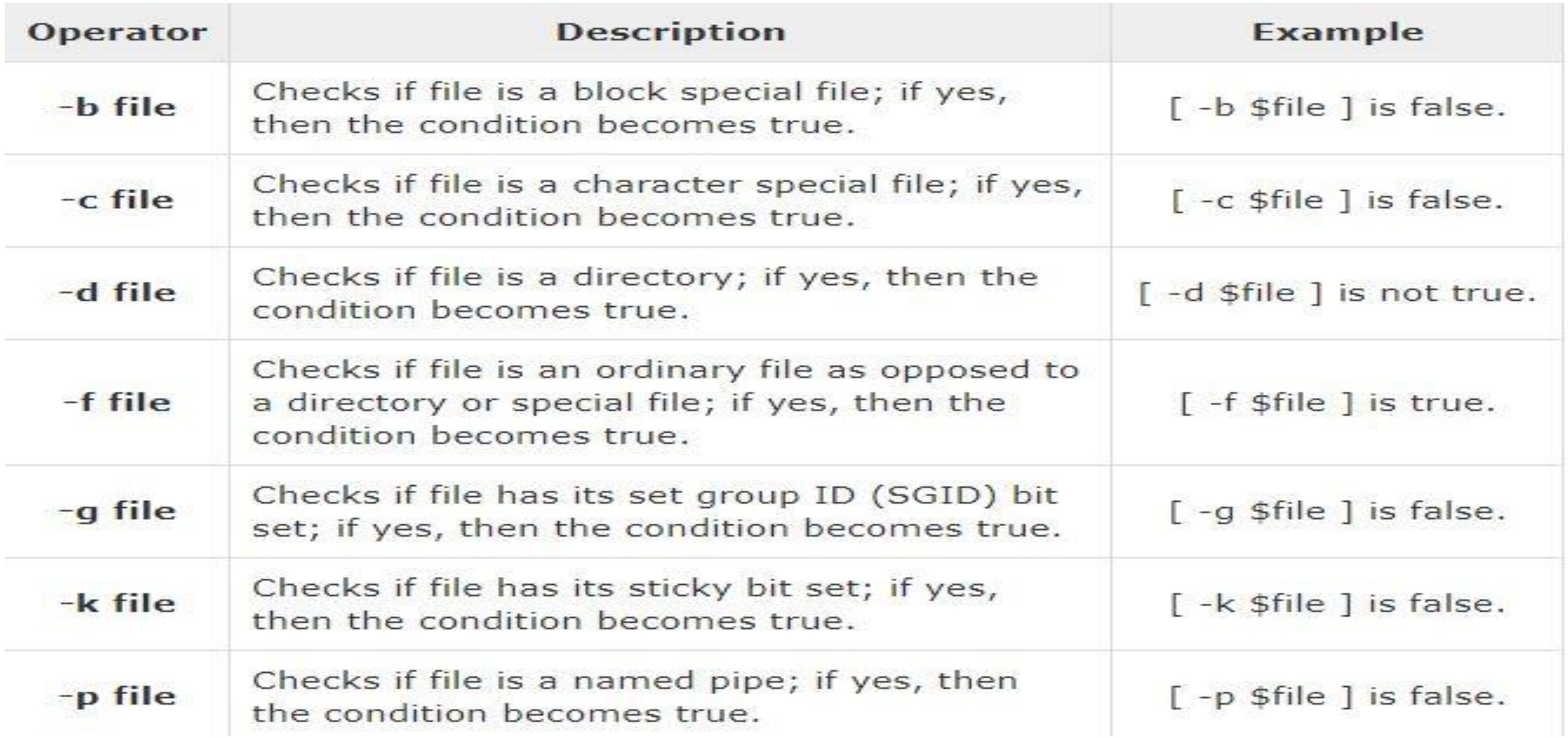
String Operators



Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

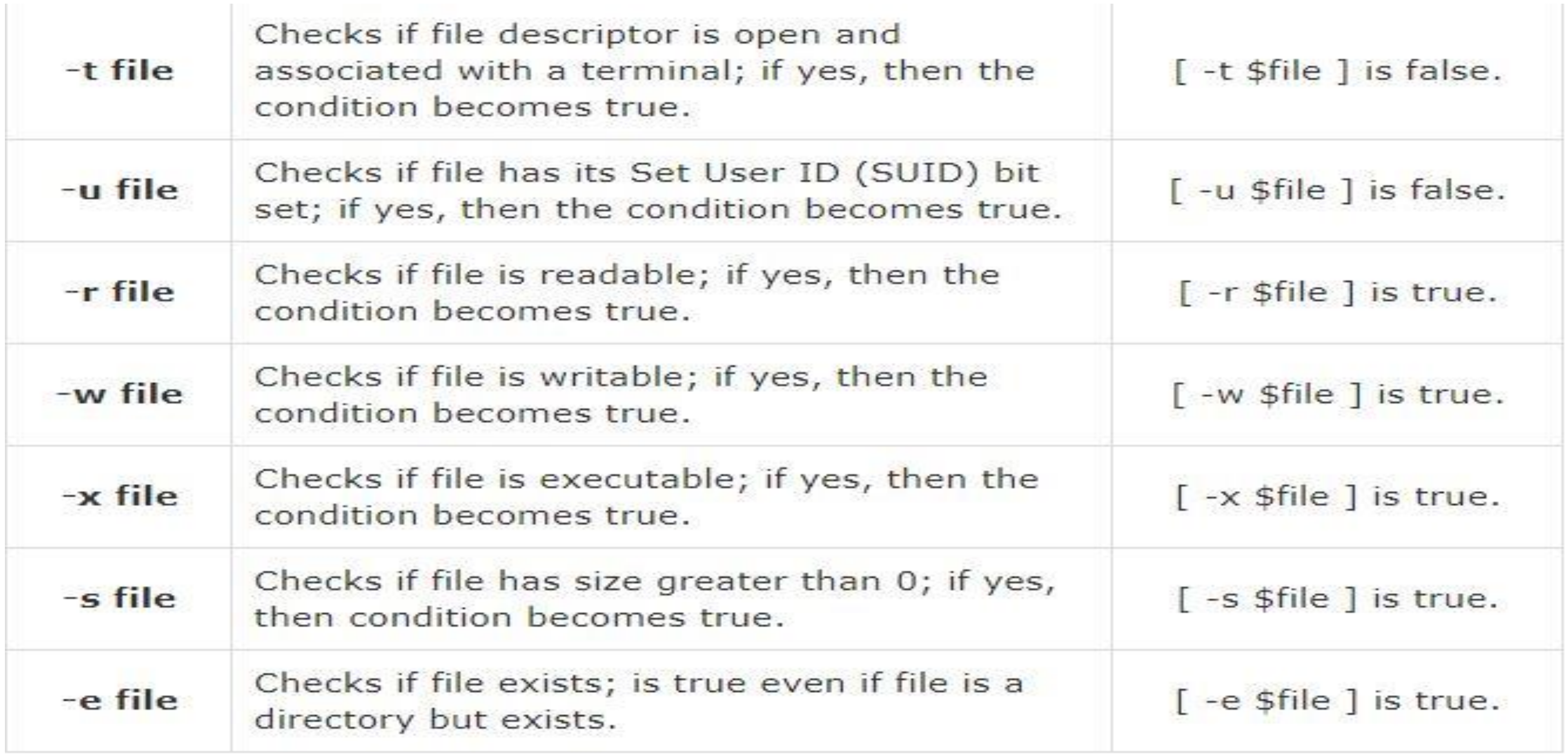
DevOps Cloud Camp
keep on practicing

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.

File Test Operators -Contd



-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.

DevOps Cloud Camp
keep on practicing

Shell Loop Types

- A loop is a powerful programming tool that enables you to execute a set of commands repeatedly.
- Types of loops available to shell programmers

- The while loop
- The for loop
- The until loop
- The IF loop

For , While and Until loops

for loop	While loop	Until loop

for i in $(ls); do
echo item: $i
done
	#!/bin/bash
	COUNTER=0
	while [$COUNTER -lt 10]; do
	echo The counter is $COUNTER
	let COUNTER=COUNTER+1
	done
		#!/bin/bash
		COUNTER=20
		until [$COUNTER -lt 10]; do
		echo COUNTER $COUNTER
		let COUNTER-=1
		done

DevOps Cloud Camp
keep on practicing

Basic IF Loop

IF ELSE	Basic IF

if [<some test>]
then
<commands>
else
<other commands>
fi
	if [<some test>]
	then
	<commands>
	fi

DevOps Cloud Camp
keep on practicing

Nested IF loop

If Elif Else	nested if statement

if [-r somefile]; then
content=$(cat somefile)
elif [-f somefile]; then
echo "The file 'somefile' exists but is not readable to the script."
else
echo "The file 'somefile' does not exist."
fi
	if [Condition]
	then
	command1
	command2
	…..
	else
	if [condition]
	then
	command1
	command2
	….
	else
	command1
	command2
	…..
	fi
	fi

Linux - Shell Substitution

What is Substitution?

- The shell performs substitution when it encounters an expression that contains one or more special characters.

Here, the printing value of the variable is substituted by its value.

Same time, **"\n"** is substituted by a new line

```
#!/bin/sh
a=10
echo -e "Value of a is $a \n"
```

$\rightarrow$

```
Value of a is 10
```

Escape sequences which can be used in echo command

Sr.No.	Escape & Description
1	\ backslash
2	**\a** alert (BEL)
3	**\b** backspace
4	**\c** suppress trailing newline
5	**\f** form feed
6	**\n** new line
7	**\r** carriage return
8	**\t** horizontal tab
9	**\v** vertical tab

Shell Functions

- Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.
- To declare a function, simply use the following syntax

```
Function_name () {
  list of commands
}
```

Creating Functions

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World"
}
# Invoke your function
Hello
```

```
$./test.sh
Hello World
```

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World $1 $2"
}
# Invoke your function
Hello Zara Ali
```

Pass Parameters to a Function

```
$./test.sh
Hello World Zara Ali
```

Nested Functions

```
#!/bin/sh
# Calling one function from another
number_one () {
   echo "This is the first function speaking..."
   number_two
}
number_two () {
   echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

```
This is the first function speaking...
This is now the second function speaking...
```

Do Hands On Labs

DevOps Cloud Camp
keep on practicing