UPES

Xebia

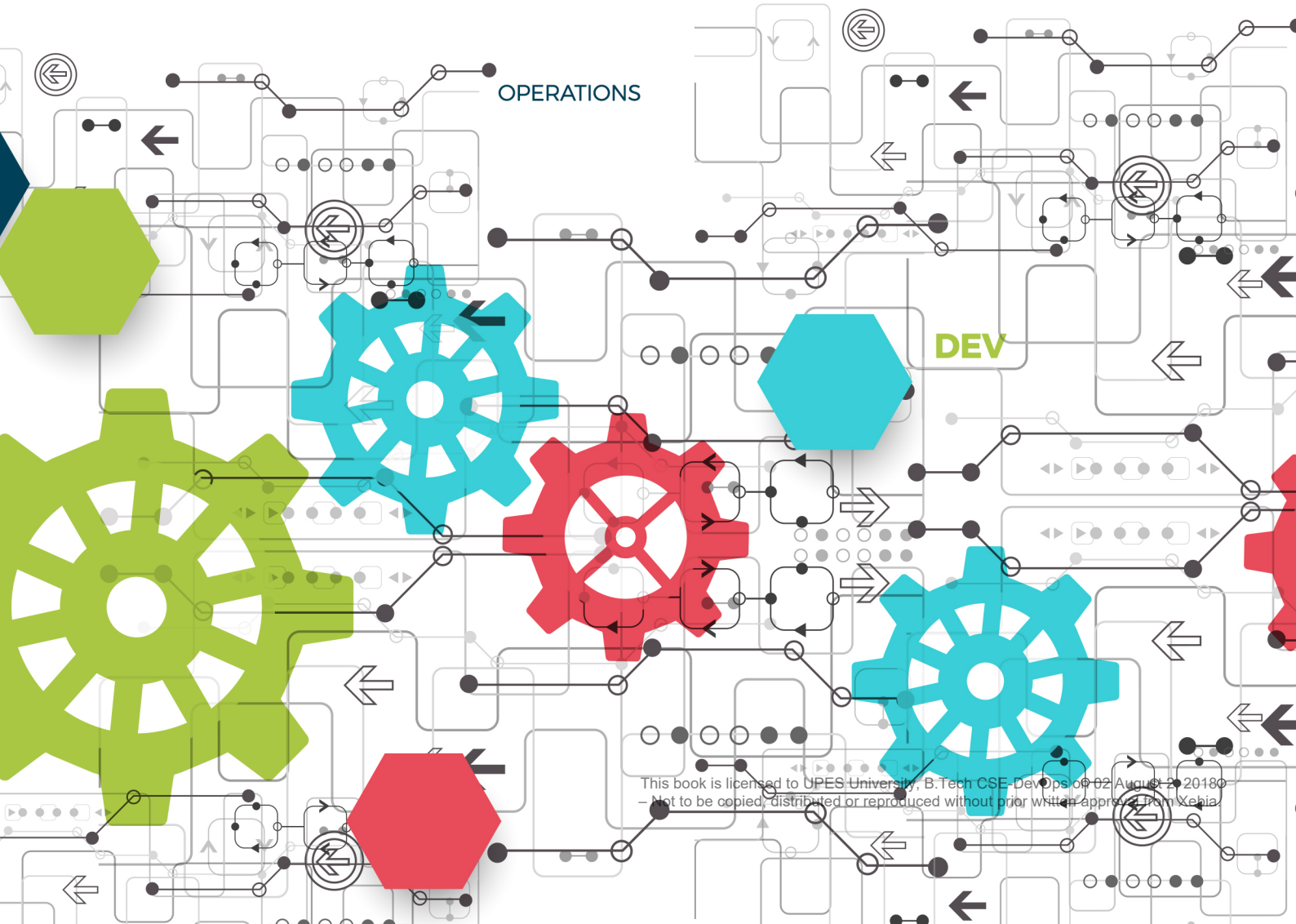
2 YEAR

B.Tech Computer Science and Engineering in DevOps

DEVOPS AUTOMATION

Semester 03 | Facilitator Handbook

Release 1.0.0



Copyright & Disclaimer

B. TECH CSE with Specialization in DevOps

Version 1.0.0

Acknowledgements

We would like to sincerely thank the experts who have contributed to and shaped B. TECH CSE with Specialization in DevOps. Version 1.0.0

SME

Rajagopalan Varadan
A tech enthusiast who loves learning and working with cutting-edge technologies like DevOps, Big Data, Data science, Machine Learning, AWS & Open stack

Course Reviewers.

Aditya Kalia | Xebia
Maneet Kaur | Xebia
Sandeep Singh Rawat | Xebia
Abhishek Srivastava | Xebia
Rohit Sharma | Xebia

Review Board Members.

Anand Sahay | Xebia

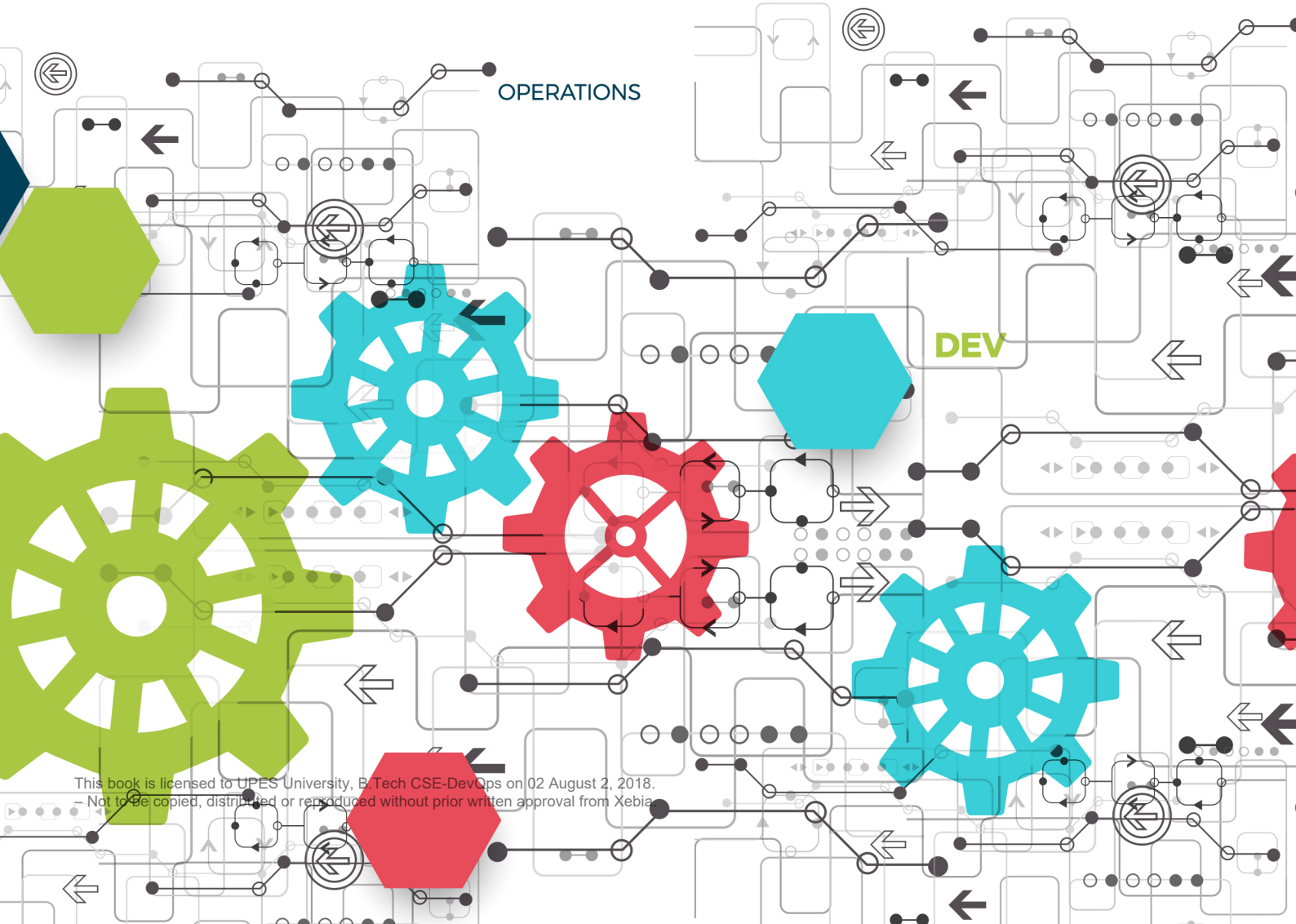
Xebia Group consists of seven specialized, interlinked companies: Xebia, Xebia Academy, XebiaLabs, StackState, GoDataDriven, Xpirit and Binx.io. With offices in Amsterdam and Hilversum (Netherlands), Paris, Delhi, Bangalore and Boston, we employ over 700 people worldwide. Our solutions address digital strategy; agile transformations; DevOps and continuous delivery; big data and data science; cloud infrastructures; agile software development; quality and test automation; and agile software security.

ODW is dedicated to provide innovative and creative solutions that contribute in growth of emerging technologies. As a learning experience provider, ODW strengths include providing unique, up to date content by combining industry best practices with leading edge technology. ODW delivers high quality solutions and services which focus on digital learning transformation.

B.Tech Computer Science and Engineering in DevOps

DEVOPS AUTOMATION

MODULE 4
Scripting Development Tasks



Contents

MODULE 4

Scripting Development Tasks

This module is the fourth module of the course and talks about Scripting Development Tasks.

Module Learning Objectives

At the end of the Module you would be able to learn the following

- Explain the basic scripting concepts such as variables, functions, etc.
- Distinguish between scripting languages (primarily on BASH and PowerShell).
- Write scripts to automate some regular tasks on their own.
- Describe and implement the best practices involved in writing scripts.

Module Topics

Let us take a quick look at the topics we will cover in this module:

1. Writing Automation Scripts
2. Best Practices for Scripting

Scripting Development Tasks

Script(s):

- A Script is a process of writing program code.
- Scripts are mostly interpreted languages and are executed on step-by-step basis manually.

- Scripts extend from a highly domain-specific language to general-purpose programming languages.

Typically, these languages are very easy to start with and they are often provided with Read-Eval-Print-Loop (REPL).

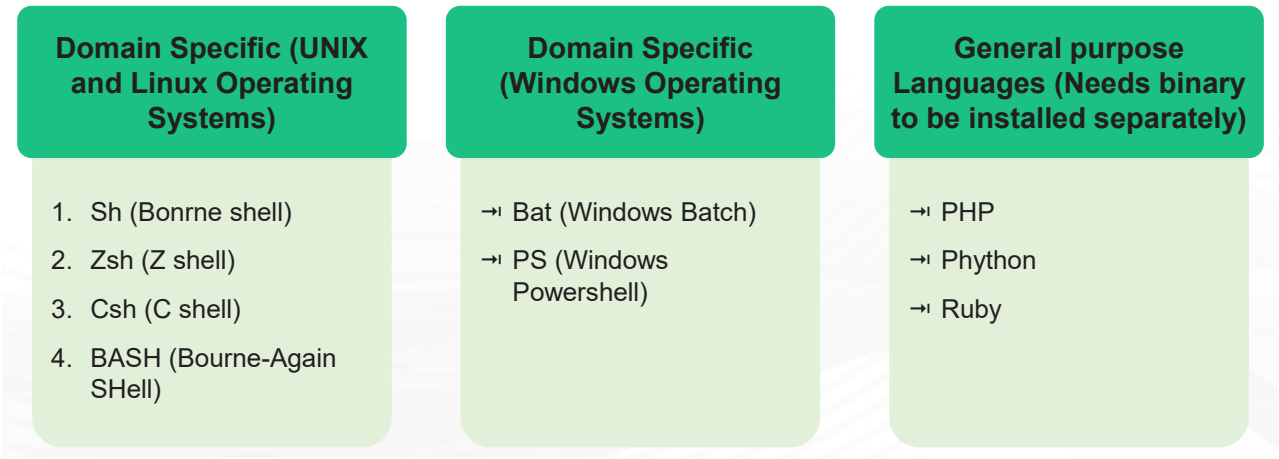
A script is just a piece of code written mostly in a domain specific language to automate a set of tasks. Remember general purpose programming languages can also be used to write scripts and hence cannot be ruled out.

Moreover, programming languages are more capable in terms of logic functionality while the scripting languages offer utilities that handle the actions specific to the domain so that we do not need to write much code in a script. Unlike some programming languages (viz. C, C++, Java, C#, etc.,), all the scripting languages are interpreted languages and it means they will be executed line by line instead of compiling the whole code to the binary language and executing them.

Since they are interpreted languages, they mostly offer a Read-Eval-Print-Loop interface (REPL) called Shell. Some of these, we might know as Bash, Csh, Zsh, etc.

Various Available Scripting Languages

The following are a few examples of Scripting languages:



As we saw in the earlier slide, the programming languages vary from a range of very highly Domain-specific language to a more General- purpose language. Choosing between them really depends on the domains we need and the functionality of the program. Some examples of languages in each range are specified in the slide.

- If you are going to write a script that deals with things specific to the Linux domain alone, then it is better to write them in Bash or any other domain specific languages since they offer to use many built-in utilities.

- In case you need a script that works across the platforms like Windows, Linux and MAC, then the general purpose languages, like python or ruby will be the best choice to use.

Most Common Scripting Languages

Linux OS

- BASH – Superset of Bourne shell
- Most commonly implemented language across all Linux distributions.
- Supports wildcard matching, piping, redirection, command substitution, variables, control structures and Loops.
- Default command shell in most Linux systems.

Windows OS

- PowerShell – A powerful framework replacing Window NT's cmd.exe
- Available from Windows 7, Windows server 2008 .
- Later can also support other operating systems with limited functionality.
- Supports variables, piping, functions, control structures, loops, try-catch.

Bash is the most common scripting language in Linux and PowerShell is preferred the most, in Windows by modern professionals. It makes sense to use them in their respective domains providing the following advantages:

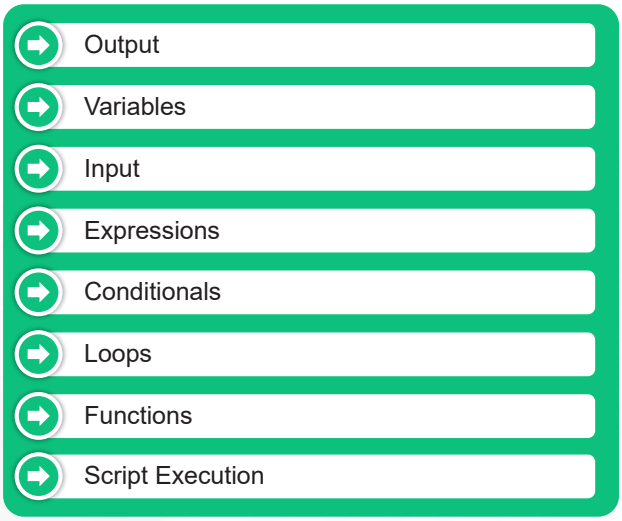
- Can get the help from the broader community.
- Get updates very frequently since much people are using them in battle-tested production environments.
- Available natively in the domain so there is no need to download source, compile them to binary and configure them.

Try to figure out, how they have evolved over the time and what makes them stand out from the other languages.

1. Writing Automation Scripts

1.1 Some Terminologies

Some of the common terminologies you should be aware of, before writing any script.



Read out the terms mentioned in the slide. These are the important concepts that one should know before writing any script. We will look at each of these in detail, in the upcoming slides.

1.1.1 Output

The text displayed on the terminal while running the script.

- **stdout** – Standard output displayed in the console
- **stderr** – All the error messages are sent to stderr

Bash:

```
# sends the text to the stdout
echo "any standard text can be here"

# sends the text to the stderr
echo "any error text can be here" >&2
```

In Linux, there are three streams and the streams responsible for the output are stdout and stderr. Each program in Linux exits with a code that can be used to identify whether the program ran successfully or not. This code at the exit of the program is called Exit Code and is used much in the lab exercises.

A successfully executed program without errors should end with the zero Exit Code whereas a program with an error should end with a non-zero Exit Code and based on the number of Exit Code, the type of the error can be identified.

- stdout, also known as the Standard Output, is the stream in which the text is displayed on the terminal when the program gets executed successfully.
- stderr, also known as the Standard Error, is the stream in which the text is displayed on the terminal when the program ends in failure.

We need to throw errors in our script wherever required so that the user of the script can identify the root causes and fix them.

1.1.2 Variables

- A placeholder to store any data on to the memory and can be accessed back in the script wherever required.
- Can be overwritten with new values and its data can be manipulated.

Bash:

```
# Assigning the text "User!" to the variable
myName="User!"

# Substituting it by enclosing in ${}
echo "Hello ${myName}"
```

A Variable is just a placeholder or a container with name labelled to it for identification. This container can contain any text or can even be empty. Once any content is added to the container, it can be used anywhere, any number of times just by referring to the name of the container.

Any text can be assigned to a Variable by defining the Variable's name first, followed by an equal symbol and then finally the text we want, enclosed in double quotes. The value of the variable can be used by Variable Substitution Operator ${variable}. Just by enclosing the variable name in this, the value of the variable gets substituted.

1.1.3 Input

- The text given to the script will be used elsewhere in the script to execute a particular action
- Generally, the given input will be assigned to a variable and its value can be accessed just by substituting the variable name.

- **stdin** – Any input received by the script is sent through stdin

```
Bash:
# Reading the text from stdin and assigning it to the variable
named "myName"
read myName
echo "Hello ${myName}"
```

The third stream, **stdin**, also known as Standard Input, is responsible for getting the values from the terminal. **read** command, arguments, environment variables, input redirection operator are some of the ways in which an input can be given to the script. Arguments and read command are the most preferred ways than the remaining.

Arguments allow us to pass the input before the actual execution of the script starts whereas **read** command gets the input whenever it gets hit in the script.

read expects its first argument to be the name of the variable. When the execution of the script hits this line, it waits till it receives a Return character from the stdin. Also, for usability sake, read returns all the stdin to stdout so that the user knows the typed text. Finally, the value from stdin is stored in the variable specified as the first argument to read command. It can be used just like any other variable.

1.1.4 Expressions

A combination of values, variables, functions, arithmetic and/or comparison operators that reduces to a primitive value.

```
Bash:
# Expressions should be placed under Double Parenthesis $((
expression ))

# Arithmetic operators ( + - * / % ** )
echo $((1 + 3 * 5 / 3))  # Prints 6

# Comparison operators ( == != < <= > >= )
echo $((1 != 8))  # Prints 1 indicating true
```

An expression is just some pieces of text that will shrink to a value when it gets evaluated. Generally, the arithmetic expressions are wrapped in double parentheses i.e. ((expression)). It is possible to use single and double brackets as well for evaluating expressions.

Bash provides us with a set of operators to do arithmetic operations and comparison operations. Comparison operators will be inevitable in writing most of the scripts since you might often be required to check whether a file exists or a variable is null etc.

1.1.5 (a) Conditional Statements – *if*

- Perform different actions based on the Boolean value of the given expression.
- Conditions are evaluated from top to bottom and as soon as an evaluation is true, that particular block of code gets executed and exits the if statement.

```
Bash:
# A basic if-elif-else statement.
# Any Boolean expression can be given to if or elif as an input.
var1 = 5
var2 = 5
if [ $var1 -gt $var2 ]; then
  echo "$var1 is greater than $var2"
elif [ $var1 -lt $var2 ]; then
  echo "$var1 is lesser than $var2"
else
  echo "$var1 is equal to $var2"
fi
```

Any language used in computation will inevitably have an if statement that allows the language to take a logical decision and do a specific set of actions defined by us. These actions are defined under the scope of the if statement and an expression that gets evaluated to a Boolean value is passed to allow the if statement to take a logical decision.

So if a given value is true, it executes the given block of code. If not, it just skips the code under its scope. If an else statement is defined, it executes the code under else block if the given value is false.

The third one, elif of the if statement, checks another given value, executes its block, if the given value is true, else it will pass on to the next elif statement or the else block if defined. “elif” and “else” blocks are optional; “elif” blocks can be added any number of times.

These statements can be nested, also called as nested If statements. Multiple conditions can be specified in a single if statement using multiple elif blocks, called as Ladder If statements.

1.1.5 (b) Conditional Statements – *Switch Case*

- Same like If condition, it performs different actions but based on the pattern matching the given input rather than the Boolean value of the expression.
- Starts with “**case**” keyword and ends with “**esac**”.
- Can contain any number of switches each having a pattern for its input.

- Each switch should begin with "**)**" and should end with "**;;**".

```
Bash:
# A string is given as input to case statement.
# Checks for the switch that matches and executes that action.
# If none is matched, "*" switch gets executed.
name="john"
case "$name" in
   john)  echo "Welcome Admin"  ;;
  alexa)  echo "Welcome User"  ;;
      *)  echo "Access Denied"  ;;
esac
```

A switch case statement is also used to take a logic decision based on the given input just like IF statement with some difference. Instead of passing in the Boolean value, we pass a string directly and the case statement compares this string with a list of regular expressions set in switches.

When the given string matches the regex of any switch, it immediately stops comparing other switches and executes the block of code defined in the currently matched switch. This same functionality can be achieved by using Nested IF statements, but the case statement will be more feasible and elegant in a script. It is the best practice to have a catch-all type regex (i.e. * as the regex) in the final switch so that if none gets matched, this block will get executed.

Remember the switch structure. It should begin with a regex followed by a closing parenthesis, the code block to be executed with a double semicolon.

1.1.6 (a) Loops – *for*

- Used to repeat an action for each item in a list or for a particular number of time.
- Defined by "**for**" keyword. Should start the loop with "**do**" keyword and end with "**done**" keyword.

```
Bash:
# Loop through the items delimited by whitespace in the "names" variable.
names="Name1 Name2 Name3"
for name in $names; do
  echo "Hello $name"
Done
# Loop through the items in the substituted sequence command.
for number in `seq 1 10`; do
  echo "I am $number"
done
```

A Loop is also a statement available in all the languages that provide the functionality of doing a defined action again and again for any number of times. A '**for loop**' is generally used to iterate for

a known number of times. For example, you want to read all files in the current folder, you would know the count of the files you have. It just needs a list of items delimited by space as an input, iterating over each item is taken care of.

do and **done** are used to define the scope of the For statement and the code within this scope will be executed for each iteration. Remember any piece of code can be used within the For statement scope. It can be an another For statement, or an IF statement or just anything that the shell supports.

A current loop can be skipped to the next loop using the **continue** command.

The **break** command is used to abruptly exit and end the loop. Generally, it makes sense to use the last two commands within a conditional statement rather than just using it directly.

1.1.6 (b) Loops – *while*

- Used to repeat an action till a condition is false.
- Defined by "**while**" keyword. Should start the loop with "**do**" keyword and end with "**done**" keyword

```
Bash:
# Evaluates whether "count" is greater than zero and if true,
enters loop.
# Print the "count" and decrease the value of count.
# Continue the same account till the count gets zero.
count="9"
while [ $count -gt 0 ]; do
  echo "$count"
  count=$[$count-1]
done
```

Same like For Loop, the While Loop is used for iterating a piece of code within its scope for a number of times. The primary difference is that, instead of looping for a particular number of times, the While Loop iterates till a given condition gets false. That is when the given condition is true, the while loop will do an iteration and again evaluates the condition. Till the condition becomes false, the while loop will keep on iterating the code block continuously with getting exhausted like humans.

The surprising part is that if the condition never gets true, then the loop will iterate till infinity, means the loop doesn't end at all. Be careful while defining any infinite loops like this. In case if you are stuck in an infinite loop, press Ctrl-C to break and exit the loop. Same like For loop, the While loop also supports the **continue** and the **break** commands.

1.1.7 Functions

- Used to define a set of actions and those actions can be executed in the script wherever needed by calling the function.

- It can optionally take inputs called parameters and can output a value.

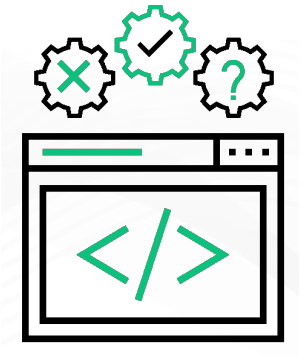
```
Bash:
# Defining the function "greet" that accepts a parameter
function greet {
  echo "Hello $1!"
}

# Calling the function "greet" with parameter "World"
greet "World"

# Calling the function "greet" with parameter "User"
greet "User"
```

A function is just a definition that contains a piece of code. This piece of code can be executed any number of times by calling the function name. This allows us to avoid rewriting the same logic again and again in multiple places of the same script.

Moreover, the functions offer a way to modify the output it returns by accepting input arguments. Also by having the conditional statements and loop statements inside the function, and by coupling them with the input arguments, any kind of functionality can be created and this can be quite handy in reusability of code. Remember, the function can write any text to the stream or it can return the text. Both of them are different ways of getting the function's output.

A function can return the value by using the return command. As soon as this command is called, the function exits immediately. The first argument passed to the return command will be the return value of the function. Also, there can be any number of return commands can be kept within the conditional statements. Returning a value from a function is completely optional.

1.1.8 Executing Scripts



- A script is a file that contains several input, output, variables, expressions, conditionals, loops and functions
- All of them together forms the logic that helps in automating tasks.
- To execute the script, it just need to be called like any other binaries.

  Ex: `/home/user/script.sh` or simply `./script.sh` if the script is in the current directory
- The script should begin with the shebang operator (#!) like `#!/bin/bash` so that the system knows which interpreter to use.
- The script files should have execution permissions.
- Usually the bash script files should end with `.sh` extension.

A script can be executed just by calling its name with its path. It can be either the absolute path or the relative path of the script. A very important thing to remember- Any script should always begin with a shebang operator i.e. the interpreter to use prefixed by the # and the ! sign.

To indicate the file as an executable script, it is generally a best practice to name them with a .sh extension. This allows the users to easily identify whether a file is a script or not. If the script file doesn't have the execution permission, calling the script will show an permission denied error message. To assign execution permissions, just run **`chmod +x script.sh`** where chmod command is the built-in tool to change file permissions, + indicates adding and x indicates the execution permission and finally the filename to which the permission is required.

1.2 Reading Arguments Passed to Script

A script takes the name of the script as the first argument ($0). All the remaining arguments will be assigned to variables ($1, $2, $3 and so on).

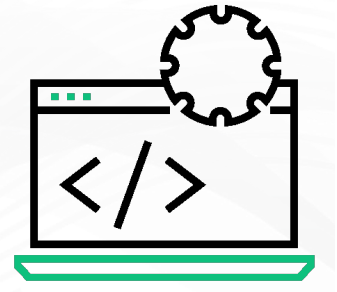
```
Bash:
In ./script.sh
#!/bin/bash
echo "File Name: $0"
echo "First Argument: $1"
echo "Second Argument: $2"
Execution:
./script.sh 1st_arg 2nd_arg
# Output
File Name: ./script.sh
First Argument: 1st_arg
Second Argument: 2nd_arg
```

An argument is another kind of input that a program can get and this input is provided while calling the script. A script can get any number of arguments, each separated by a space. It is generally good to limit the arguments not more than five. Else calling the script becomes difficult. All the arguments will be available for digits 0 to N prefixed with $ sign.

If you want to pass a text having the space as a single argument, enclose the text within the double quotes. Provided arguments can be popped out by using the **`shift`** command.

1.3 Task Scheduling Using Cron

Cron



- A task scheduler in Linux that runs as a daemon.
- Helps in running a program at the specified intervals.
- "**`crontab -l`**" lists all the entries and "**`crontab -r`**" removes the entries.
- Simply typing "**`crontab -e`**" opens a editor where the job can be scheduled.

Cron Expression:

- `<second> <minute> <hour> <day-of-month> <month> <day-of-week> <year> <command>`

Cron is a task scheduler application in Linux operating system that continuously runs as a daemon process. A cron expression is a string having a structure where the intervals, script and their arguments can be specified by separating them with spaces. `crontab` is a command that allows us to add, remove and list available cron expressions by passing different options.

1.4 Basic Linux Commands

The following table describes the basic Linux commands.

COMMAND	OPERATION
pwd	Prints the current working directory
ls	Lists all the files and folders
cd	Change the current directory
touch	Update the timestamp of a file
cat	Prints the content of a file
mkdir	Creates a new directory
rmdir	Removes an existing directory
cp	Copy the specified files to the given location
mv	Move the specified files to the given location
rm	Deletes the specified files
man	Show help

A Linux command is just a binary executable file located somewhere usually /bin directory or /usr/local/bin directory specified in the Path environment variable at /etc/environment. Each one of them is written in a programming language like C and then compiled to a binary. These commands will be commonly used in day to day tasks of a typical IT administrator.

Also, we can say that it will be quite hard to work on Linux terminal without these basic commands. Go through them one by one, keep them in your memory by any means as you will need them for even a very basic action.

The following table describes the basic Linux commands.

COMMAND	OPERATION
echo	Prints the given text to the stdout
sudo	Become the root user
df	Prints the Available disk space of each partitions
du	Prints the Disk Usage of a file
apt	Helps to install new software in the system
chmod	Change the filesystem permissions of a file
chown	Change the ownership of a file
vim	Edit the contents of a file
find	Search and locate the files matching the criteria
clear	Clears the console

The commands listed in this slide are important but not mandatory if you are just going to start off. However, it is very crucial to know these commands when you are writing a script. Else you will end up in reinvent the wheel when it is already available.

1.5 Redirection Operator

"<" – Input redirection operator

Gets the name of the file from its right side and passes the content of the file to the command specified on its left side.

">" – Output redirection operator

Get the output from the command on its left side and passes the captured output to the files on its right side.

Example: ls > files.txt

Explanation:

The 'ls' command prints all the files to the stdout

The output redirection operator will get the stdout value and sends it to the 'files.txt' file.

The Redirection operators are typically used to write and stream from one of the streams in the Linux Operating System. For example, input redirection operator is capable of getting a file's content and write the content to the stdin stream.

Likewise, the Output Redirection operator can get the content from the stdout or stderr stream and write them to a file. This allows us to manipulate the data between streams across different programs or scripts.

1.6 Piping Operator

"|" – Piping operator

Send the output from one program as an input to another program

Example: ls –l | head –5

Explanation:

- The 'ls' command prints all the files to the stdout
- The piping operator will get the stdout value and sends it to the head command
- The head command prints only the first 5 lines of the output.

A piping operator can is used to fetch the content of the stdout of one program and pass the content to the stdin of another program. This can be done without piping operator by just using standard stream operators. However piping operator provides a very convenient method of handling these kinds of operations and will help in maintaining the code elegant. Also, this allows chaining of multiple commands which is quite difficult with standard redirection operators.

1.7 (a) Sample Program - 1

```
#!/bin/bash
if [ "$#" -eq 0 ]; then
  echo "ERROR: Files/Folders to search & replace is not specified" >&2;
  exit 1;
fi
item_paths=""
for i in $@; do
  item_path=$(realpath $i)
  if [ -f "$item_path" ]; then  item_paths+="$item_path "
  elif [ -d "$item_path" ]; then  item_paths+="$item_path/* "
  else   echo "ERROR: The given path \"$item_path\" is not a suitable type" >&2;
exit 1;
  fi
done
read -e -p "Search For: " search_text
read -e -p "Replace With: " replace_text
sed -i "s/$search_text/$replace_text/g" $(grep -lz "$search_text" $item_paths)
```

Refer the scenario 2 of "Automation Saves Time and Efforts" of Module 2.

This sample program is the solution for the second scenario of the "Scenarios where automation prevents errors" topic in the second module. Here, we have a script that will help us to search and replace a text across multiple files. We just need to pass the files to include, the text to search for and the text to replace with. For an explanation of the program, refer to the notes of this particular scenario in the second module.

1.7 (b) Sample Program - 2

```
#!/bin/bash

read -e -p "Log Directory: " log_directory

read -e -p "File Extension: " extension

read -e -p "Backup Directory: " backup_directory

tar czf archive.tar.gz $(find $log_directory -name "*.$extension")

mv archive.tar.gz $backup_directory/$(date +%F).tar.gz

rm $(find $log_directory -name "*.$extension")
```

Refer the scenario 1 of "Automation Saves Time and Efforts" of Module 2.

This sample program is the solution for the first scenario of the "Scenarios where automation saves time and effort" topic in the second module. This script helps in automatically archiving all the log files and moving those archives to a backup directory and finally deleting the original log files.

It expects the location of the log directory, the extension of the log files and the backup directory where the archives should be kept. For an explanation of the program, refer to the notes of this particular scenario in the second module.

1.7 (c) Sample Program - 3

```
#!/bin/bash
for files; do
  lines="$(wc -l < $files | sed 's/ //g')"
  characters="$(wc -c < $files | sed 's/ //g')"
  owner="$(ls -ld $files | awk '{print $3}')"
  linecount="1"
  echo "----------------------------------------------------------------"
  echo "File $files ($lines lines, $characters characters, owned by $owner):"
  echo "----------------------------------------------------------------"
  while IFS= read -r line; do
  printf "%02d" $linecount
  echo ": $line"
  linecount="$(( $linecount + 1 ))"
  done < $files

  echo "----------------------------------------------------------------"
  echo -e "\n\n"
done | less
```

Refer the scenario 11 of "Automation Saves Time and Efforts" of Module 2.

This sample program is the solution for the eleventh scenario of the “Scenarios where automation saves time and effort” topic in the second module of third semester. It just prints the content of the files in a formatted manner along with additional information. It requires the files to be read as the arguments of the script.

For an explanation of the program, refer to the notes of this particular scenario in the second module.

What did you Grasp?

1. Which of the following commands reads the user input and assigns it to a variable?

   A) read
   B) get
   C) declare
   D) Set

2. Choose the correct method for appending the text “Hello World” in “/tmp/myfile” file.

   A) echo *“Hello World”* > */tmp/myfile*
   B) echo *“Hello World”* >> */tmp/myfile*
   C) echo *“Hello World”* | */tmp/somefile*
   D) /tmp/myfile < echo “Hello World”

2. Best Practices for Scripting

2.1 Make use of Shell’s Built-In Options

- “`set`” is a built-in utility that allows to modify or display the positional arguments given to the shell. “`set -o OPTION_NAME`” allows us to set the options to the shell. In our case, we need to logically set or unset the bash shell’s options.

- Make use of the following “set” commands wherever applicable in our script.

  `set -o errexit` - Exit when any command fails. Alias: `set -e`

  `set -o nounset` - Exit when undeclared variables are referred. Alias: `set -u`

  `set -o xtrace` - Display the command that gets executed. Alias: `set -x`

  `set –o pipefail` - Exit when any piped command gets failed.

- By setting these options separately instead of passing directly in the shebang where we specify the interpreter, We can logically control whether set any of them.

  Example: `[[ "${DEBUG}" == 'true' ]] && set -o xtrace`

  This sets the xtrace only when the DEBUG environment variable is set to true.

‘*set*’ is itself a built-in command available on linux that helps to change the arguments provided to the script. It is absolutely possible to provide the shell with those arguments without the set utility. But it can’t be changed or controlled logically, once the execution of the script is started.

To have a control over it, we use the ‘set‘ tool in our scripts. So by setting relevant environment variables and enabling or disabling options in set with those environment variables will provide a very nice usability of the program.

2.2 Naming Conventions Save Time and Effort

Do’s	Don’t’s
⇥ Variable or function names should have only letters, numbers and optionally underscore character.	⇥ It should not contain any other special character and even spaces.
⇥ It is better to begin the name with a letter and use underscore to separate the words in them instead of spaces.	⇥ Do not begin the variable name with numbers.
⇥ It is okay to begin your function name with an underscore character if the name conflicts with any other system tool.	⇥ Do not use a complete uppercase text as the name unless it is an environment variable or globally exported variable.
⇥ Give intuitive names to variables.	
⇥ Conform to the language conventions and even the naming of the scripts are important too.	
⇥ Prepare a style guide before starting to write the code.	

As shown in the slide, variable or function names should have only letters, numbers and optionally underscore character.

- It is better to begin the name with a letter and use underscore to separate the words in them instead of spaces. Numbers can be used wherever it is appropriate.

- It is okay to begin your function name with an underscore character if the name conflicts with any other system tool.
- Give intuitive names to variables so that it will be self- explanatory of what it contains.
- Conform to the language conventions and even the naming of the scripts are important too.
- Prepare a style guide before starting to write the code so that all the people working on the it can easily understand and resume quickly with their work.
- It should not contain any other special character and even spaces.
- Do not use a complete uppercase text as the name unless it is an environment variable or globally exported variable.

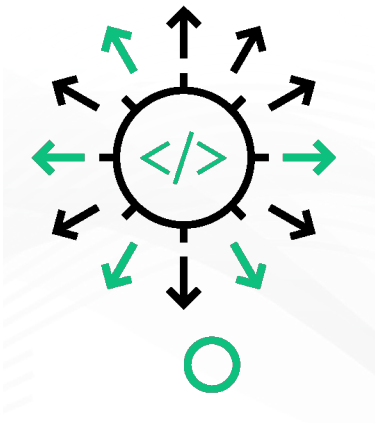
2.3 Annotations Make the Logic Clean

Annotations:

- Are similar to access modifiers in general programming languages.
- Define the scope and behavior of the variables that are annotated.
    - Unlike general programming languages, bash does not support a complete list of annotations yet it supports to basic forms.

      local – Defines a variable to available only within the scope in which the variable is defined.

      readonly – Defines the variable as an immutable object. Does not allow to reassign any new values to the variable.



- Allows us to quickly debug and write a scalable complex code neatly.

Annotations are nothing but access modifiers that define the scope of the variable and the operations supported on it. Scope refers where the defined variable can be accessed and where it cannot be accessed. **Example:** Your property can be accessed by your family but cannot be accessed by any stranger. It is similar to a local variable that can be accessed within the defined scope and is not reachable beyond the scope. A global variable is like a public property that can be accessed by anyone.

Assigning and referring are the basic operations available on a variable. Read-only annotation will allow the variable to be assigned with a value only on its initialization stage. Referring the value of a variable can be done any number of times.

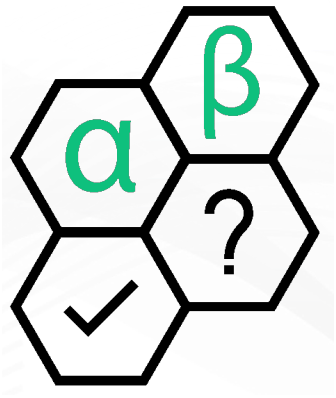
As much as it is a good thing to annotate the variables, it also depends on the place and the role of the variable we define. Take them into consideration as well. **Example:** A normal variable is like writing with a pencil on a paper. The written text can be erased and rewritten any number of times. Whereas the read-only variable is like writing with a pen on the paper. Can be written only once.

2.4 Command Substitution

- Command Substitution:
- One of the most frequently used feature in which the stdout of any commands can be assigned to a variable or substituted wherever it is required.
- Bash allows us to do this in two ways.

  `$(COMMAND)` – The modern way of substitution, that encloses the command with parenthesis and a prefix dollar sign. It can be nested without any additional actions.

  `` `COMMAND` `` – The traditional way of substituting is by enclosing the commands with back ticks. It is not preferred due to less readability since they can be easily confused with single quotes. Also the nested substitution needs the back tick to be escaped with '\' character.



- Always enclose the command substitution within double quotes to ensure the whitespaces returned from the command doesn't cause any trouble.

  Example: `echo "Hi, I am $(whoami)"`

A command substitution is nothing but acquiring the stdout of a shell command and assigning it to a variable or directly substituting it in the place where it is called. It is quite impossible to write complex shell scripts without command substitutions. It opens up a lot of possibilities to make use of utilities installed in Linux other than native shell commands.

- Bash allows us to do this in two ways:

  `$(COMMAND)` – The modern way of the substation that encloses the command with parenthesis and a prefix dollar sign. It can be nested without any additional actions.

  `` `COMMAND` `` – The traditional way of substituting is by enclosing the commands with back ticks. It is not preferred due to less readability since they can be easily confused with single quotes. Also, the nested substitution needs the back tick to be escaped with '\' character.

- Always enclose the command substitution within double quotes to ensure the whitespaces returned from the command doesn't cause any trouble.

  Example: echo "Hi, I am $(whoami)"

2.4 Always Begin with a Shebang

Shebang (also called hash-bang):

- Should be the first line on any shell script.
- The system uses the shebang to know, which binary to use as the interpreter for this script.
- The following shebang targets the bash as the interpreter to use for the script.

  Example: #!/bin/bash
- Though the options to the Interpreter can be passed like #!/bin/bash –x -e, it is discouraged. Because, If the script is ran explicitly executed by an interpreter, then the specified operations in the shebang will be simply ignored and will end in inconsistencies.
- For the sake of portability, using #!/usr/bin/env bash is more preferred than #!/bin/bash. This ensures all the Linux operating systems will be able to fetch the right interpreter.
- Make sure only the shebang is present on the first line of the script and nothing else. Start your script after two lines below.

A shebang operator is the one that forces the shell to use a specific interpreter to execute the script. If shebang operator is not provided, then the current shell will just use itself to execute the script which is not feasible at all times, at all places. The script will be developed irrespective of the current shell used and one cannot assume that the current shell will be used in all places. One common way is to directly specify the path of the interpreter to use in the shebang.

2.5 Variable Substitution

- In Bash, Variables should be accessed using their name prefixed by the $ sign. This way of doing is not encouraged any more. Instead the variables should be accessed by enclosing the variable name within braces and should be prefixed with the $ sign.

  `$VARIABLE` – Not recommended. It has a potential drawback as given below.

  Example: echo "`Hi, I am $NAME_app`" will search for the variable "NAME_app" instead of just "Name".

  `${VARIABLE}` – Most recommended safe way of substituting the variable value. It also allows some additional benefits like fallback values `${name:-default_name}` or string replacement `${name//search_text/replace_text},` etc.,
- No spaces should used between equal symbol on assigning to a variable.
- General magic variables i.e. variables containing current file name, base name, directory, etc. at the top of the script since they are most often used here and there throughout the script.

Variable substitution is just using the value of a variable in places wherever it is required. Just like command substitution, It can be accessed in two ways. By using $name, but it doesn't define the exact name of the variable. It ends in the ambiguity of locating the variable. Another method ${name} explicitly defines the name of the variable and makes it clean. Besides, the second method allows a fallback value if the variable is unset and also allows to do a search and replace action on the value of the variable and then substitute it.

2.6 Conditionals

- A single square bracket has more portability than a double square bracket but its functionalities are limited.
- Expression in a single bracket should be properly escaped and enclosed.

  Ex: `[ "$var" \< a ]`

  whereas the double bracket does that automatically.

  Ex: `[[ $var < a ]]`
- Though the POSIX standards have not yet updated to include double brackets, it is available natively on all Linux distributions.
- The evaluation of the double bracketed expression can be trusted more than the single bracketed ones.
- In places where portability is not significant, double bracketed expressions are recommended to be used.

A conditional is just an evaluated expression that can be passed to conditional statements like If, While and Case statements. An expression placed between the double brackets or single brackets gets the expression evaluated with some differences. The single bracket evaluation is a primitive type where the variables should be substituted with text and the comparison operators should be properly escaped. Whereas the double bracket evaluation can directly process the variables and comparison operations without any further changes. The POSIX standard doesn't cover the double bracket. However, it is being extensively used in most of the common Linux distributions.

2.7 Avoiding Temporary Files

- At some places we may be required to pass a file instead of plain text to some utilities.

  Example: The `diff` command expects the input to be file rather than the text itself.
- An old-school technique is to write the text we have to a temporary file (usually in the /tmp directory) and then pass that file as the input to the utility.
- It has its own drawback like the conflicts that can occur in writing the temporary file, the permission for write on the temporary directory and these can be completely skipped.

- The `<(echo $variable)` operator help us to pass any text in our variable or text acquired by a command substitution as a file input to the utility we want.
- Thus passing the content as file input can be handled by the bash itself and this is recommended than writing to a temporary file.

A temporary file is nothing but a file that is used only for a very shorter period of time by a program and nothing else. This method of doing has some potential errors that add an overhead of handling the edge cases to make our script more reliable and robust. Alternatively, using the <(some text) operator, the burden is taken care of by the shell itself and we can focus on the actual logic of our script.

2.8 Regular Expressions

- Regular Expressions (often called Regex) is a sequence of characters that defines a pattern to search the text.
- Using the regex to filter the text will help in locating the exact matches than matching all the nearby ones. Using regex is recommended to match or capture the words than manually hardcoding the list of patterns.
- Double bracketed expressions support regex whereas the single bracketed expressions doesn't. Regex in conditionals will be a lot helpful on a typical shell script.
- Regex must not be quoted if they are being used inside the double bracketed expression.
- Case statement also supports regex based string matching.
- Not all the utilities supports regex in general. Double check the utility's documentation before passing the regex as input.

A regular expression is a text which is a pattern that matches exactly the text we need from a bunch of paragraphs. It is most common in programming languages to fetch the required text by using regex strings. However, most of the Linux utilities such as find, grep, ls, sed, awk, etc., support regular expressions to filter the matching text or file from the given input. Especially the utilities like awk and sed help to do a search and replace the operations using regex which is very highly useful.

2.9 Debugging

- Do not hardcode any print commands or write to the file just for the purpose of debugging.
- Use the `xtrace` or "`-x`" option of the set to print the command executed and the output provided by the command. It will even display the commands executed on the sub-shell and indicates them by prefixing the command with a double '+' character.
- The "`-n`" argument of the bash allows to run the syntax check of your bash script instead of executing the script.
- Similar to xtrace, "`-v`" or `verbose` option will print the commands written in the current script.

- If your script is a long running one, then have multiple log levels in it and write them to a log file in the standard log directory. Some standard log levels are `OFF, FATAL, ERROR, WARN, INFO, DEBUG, TRACE` and `ALL.`
- Allow users to run the script with various log levels so that even the complex scripts can be easily troubleshooted.

Debugging in literal means removing errors. It is normal to have multiple errors while writing the program for the first time. But before using this program for some real-life scenarios, it should not have any faults or surprising fatal errors.

“A well-diagnosed disease is half cured”. In our terms, “A well-defined problem is half fixed”.

The process of the debugging helps to locate and correct errors. By going through the command execution one by one, it is far easier to find mistakes. Providing information on various log levels will help the people using your script to understand the errors and take quick action to fix them.

What did you Grasp?

1. Which of the following is a valid annotation available in bash?

   A) static
   B) private
   C) readonly
   D) Global

2. Choose the valid variable substitution from the following?

   A) $(variable)
   B) `variable`
   C) ${variable}
   D) #!/variable

In a nutshell, we learnt:

1. A script is the program designed for a specific runtime to automate the basic tasks.
2. Bash is the most common scripting language in Linux and Powershell is the most preferred in Windows.
3. stdout, stderr and stdin are the three streams available in Linux.
4. All shell scripts must begin with a shebang.
5. All the shell commands can be used in the script.
6. Third party tools can be used in script just like using them in the terminal.

Release Notes

B. TECH CSE with Specialization in DevOps

Semester three -Year 02

Release Components.
Facilitator Guide, Facilitator Course Presentations, Student Guide, Mock exams and relevant lab guide.

Current Release Version.
1.0.0

Current Release Date.
2 July 2018

Course Description.
Xebia, has been recognized as a leader in DevOps by Gartner and Forrester and this course is created by Xebia to equip students with set of practices, methodologies and tools that emphasizes the collaboration and communication of both software developers and other information-technology (IT) professionals while automating the process of software delivery and infrastructure changes.

The language used in this course is US English. Our sources of reference for grammar, syntax, and mechanics are from The Chicago Manual of Style, The American Heritage Dictionary, and the Microsoft Manual of Style for Technical Publications.

Bugs reported	Not applicable for version 1.0.0
Next planned release	Version 2.0.0 Feb 2019