Linux consists of some major parts such as kernel, shell and the GUI (Gnome, KDE & so on).

* The shell translates our commands and sends them to the system. Every shell has its own features, and some of the popular ones are

- Sh shell: this is called Bourne shell.
- Bash shell: Also called Bourne again shell, this is very popular and compatible with sh shell scripts, so you can run your sh scripts without changing them.
- Ksh shell: Also called Korn shell, it is compatible with sh and bash.

what is bash Scripting?

* The basic idea of bash scripting is to execute multiple commands to automate a specific job.

* we can run multiple commands from the shell by separating them with semi colons (;):

ls ; pwd

The first command runs, followed by result of second command

Every keyword we type in bash scripting is actually a linux binary (program), even the 'if' statement or 'else' or 'while' loops. All are linux executables.

* we can say that the shell is glue that binds these commands together.

command "type"

Bash's built-in 'type' command will display the type of command for a given word entered at the command line.

demonstration of simple use 'type'

```
$ type ls
ls is aliased to 'ls --color=auto'
```

we can extend this to display all the matches for the given command:

```
$ type -a ls
ls is aliased to 'ls --color=auto'
ls is /bin/ls
```

If we need to just type in the o/p, we can use -t option.

```
$type -t ls
alias
```

It is also used to identify shell keywords such as if, and case.

```
$type ls quote pwd do id
```

o/p: ~~type~~ ls is aliased to 'ls --color=auto'

quote is a function

```
quote()
{
    local quoted = ${1//\'/\'\\\'\'};
    printf "'%s'" "$quoted"
}
```

pwd is a shell builtin

do is a shell keyword

id is /usr/bin/id

Command "PATH"

Linux will check for executables in the PATH environment only when the full or relative path to the program is supplied.

```
$ export PATH = $PATH:
```

This appends the current directory to the value of the

PATH variable: each item in the PATH is separated using a colon.

CREATING & EXECUTING SCRIPTS.

we will begin by creating a hello world script.

```
#!/bin/bash
echo "Hello world"
exit 0
```

#!/bin/bash: Normally, this is always the first line of the script and is known as the shebang. It starts with a comment, but the system still uses this line.
A comment in shell script has # symbol.
The shebang tells the interpreter of the system to execute the script.

echo: the echo command is used to write a standard output, STDOUT;

exit 0: The exit command is built-in shell, and is used to leave or exit the script.
The exit code is supplied as an argument.

A value of anything other than 0 will indicate some type of error in the script's execution.

EXECUTING THE SCRIPT

Even if the script is saved in our PATH environment, it still will not execute as a standalone script. we will have to assign and execute permissions for the file, as needed.

For example, if we run file directly with bash then we will get the correct o/p but it is directly not a long-term sol^{n}. But if we want to make running the script easy from any location without typing full path, the we need to give permissions.

Example directly!-

```
$ bash $HOME/bin/hello1.sh
```

Adding permissions

```
$chmod +x $HOME/bin/hello1.sh
```

Now we can run the script simply as follows

```
$ hello.sh
```

Checking the exit Status

Consider an example as follows

* $command1 || command2.

So, here command2 is executed only if command1 fails in some way. To be specific, command2 will run if command1 exits with a status code other than 0.

* $command1 && command2

Here command2 executes only if command1 succeeds and issues an exit code of 0.

$? variable

To read the exit code explicitly from our script, we can use $? variable

Ex: $ hello1.sh
$ echo $?

Running the script with arguments

we can run the script with arguments. However if the script does not make use of arguments, then they will be silently ignored.

Ex: hello1.sh (fred) ← argument.

o/p :- Hello world

[hello1.sh → echo "Hello world"]

↓

Here there is no space for argument. That's why the o/p is just Hello world. & the argument is ignored.

Argument Identifier	Description.
$0	the name of the script itself, which is often used in usage statements.
$1	A positional argument, which is the first argument passed to the script.
${10}	where 2 or more digits are needed to represent the argument position. Brace brackets are used to delimit the variable name from any other content. single value digits are expected.
$#	The argument count is especially useful when we need to set the amount of arguments needed for correct script execution.
$*	Refers to all arguments

Let's see the examples for above topic.

ex Script : Hello2.sh.

```
#!/bin/bash
echo "Hello $1"
exit 0
```

So, let's execute this by providing the argument.

I/p : Hello2.sh India

O/p :- Hello India [∴ since India is the 1st argument passed]

ex script : Hello3.sh

```
#!/bin/bash
echo "Hello $*"
exit 0
```

① I/p : Hello3.sh I love my India

O/p : Hello I love my India [∴ $* prints all the passed arguments]

② I/p : echo "you are using $0"

O/p : you are using /home/gauri/bin/hello2.sh

The Importance of correct Quotes.

It is very important to understand the quoting mechanisms available in bash

If we use double quotes

```
I/P :- echo "Hello $1"
O/P :  Hello India
```

[∵ we have seen this hello2.sh]

If we use single quotes, then the content which is there inside the single quotes is printed. No matter what the arguments are.

```
I/P : echo 'Hello $1'
O/P :- Hello $1
```

* normally if we write something followed by $ then the system considers it as variable, but if we really want $ in our o/p then we can do it by putting back slash (\) infront of it. (\$)

for example " gani earns \$4 "

```
o/p : gani earns $4
```

printing the script name

6 to print the script name we can use $0

Ex: echo "you are using $0"

O/p: you are using /home/gani/bin/hello2.sh

But, if we prefer not to print the path & only the name of script, then we can use basename command.

Ex: I/p: echo "you are using $(basename $0)"

Here, $(....) syntax is used to evaluate the o/p of the inner command.

we will first run basename $0 and feed the result into an unnamed variable represented by $.

so, the o/p is

you are using hello2.sh.

Declaring Variables:

There are two kinds of variables you can declare in your script.

* user-defined variables
* Environment variables.

user-defined variable

To declare a variable, just type the name and set its value using equals sign (=).

Ex:
```
#!/bin/bash
name = "gani"
age = 24
echo $name
echo $age.
```

So, to print the variables value, we should use $.

* there should be no spaces b/w variable name and equal sign (or) b/w equal sign & value.

If there is space then it will show error.

Arrays

Arrays can hold multiple values.

Syntax : myarr = { one two three }.

Ex :

myarr = { one two three four }

→ To access specific element, then we should specify ind

echo ${myarr[1]} # prints two which is the second element.

echo ${myarr[*]} # prints all the elements

→ To remove specific element, use 'unset' command.

unset myarr{1} # This removes 2nd element

unset myarr # this removes all elements

Environment Variables!

So far we have used some variables which we haven't declared. Such as $BASH_VERSION, $HOME, $PATH, $USER

So, where did they come from?

These variables are defined by shell and these are called environment variables

There are many environment variables. If we want to list them, we can use printenv command.

Variable Scope

If we define a variable in Hello1.sh and try to print it in Hello2.sh then it will not show anything. Bcoz the variable's scope is only limited to the process that creates it.

However, we can do this by using the export command

Ex:

Script 1

```
#!/bin/bash
name = "gani"
export name
./script2.sh
echo $name
```

Script 2

```
#!/bin/bash
name = "Reddy"
echo $name
```

o/p: Reddy

gani

Command Substitution

Command substitution means storing the o/p of a command in a variable.

There are 2 ways to do this.

① we will use two backticks

```
#!/bin/bash
cur_dir = `pwd`
echo $cur_dir
```

② By using dollar sigh, like this $( )

```
#!/bin/bash
cur_dir = $(pwd)
echo $cur_dir
```

Debugging your Scripts.

Bash provides two options for us -v and -x.

If we want to look at the verbose o/p from our script & detailed information about the way the script is evaluated line by line, we can use the -v option.

Ex: bash -v $HOME/bin/hello2.sh fred

we can see how each element of the embedded basename command is processed.

The first step is removing the quotes and then the parentheses.

```
O/P:  #!/bin/bash
      echo "you are using $(basename $0)"

      basename $0)"
      basename $0)
      basename $0
      you are using hello2.sh
      echo "Hello $*"
      Hello fred
      exit 0
```

The -x option, which displays the commands as they are executed, is more commonly used.

```
I/P:- bash -x $HOME/bin/hello2.sh gani
      ++ basename /home/pi/bin/hello2.sh
      + echo 'you are using hello2.sh'
      you are using hello2.sh
      + echo 'Hello gani'
      + exit 0
```

Using echo with options

The basic use of echo that we have seen so far will produce a o/p and a newline. If we do not supply any text string to print, echo will print only the new line to STDOUT.

If we don't want to transfer the control to the newline then we can do this in 2 ways.

① $ echo -n "gani"

② $ echo -e "gani\c"

In the 1st, -n option is used to suppress the line feed.

In the 2nd, -e option which allows escape sequences to be added to the text string. To continue on the same line, we use \c as the escape sequence.

Basic Script Using read

echo is used to prompt something and read is used to get I/p from the command.

```
#! /bin/bash
echo -n "Hello $(basename $0)! May I ask your name: "
read
echo "Hello $REPLY"
exit 0
```

Script Comments

Anything after # is a command and is not evaluated by the script.

* The shebang, #!/bin/bash is primarily a command and is not evaluated by shell, But the shell running the script reads the whole shebang, so it knows which command interpreter to hand the script over to.

* Shell script does not have the notion of multiline comments.

Enhancing Scripts with read prompts.

The read command with -p option can perform both actions like [illegible] prompting & taking I/p.

Ex: read -p "Enter your name! " name.

the syntax is,

read -p <prompt> <variable name>

Even if we do not supply the last argument, we can still store the user's response, but this time in the 'REPLY' variable.

Limiting the number of entered characters:

Ex: read -n1 -p "press any key to exit"

Here, if we use integer 1 followed by -n then it is saying that only one character is allowed before continuing.

Ex 2: read -n3 -p "press any 3 keys"

since there is 3 followed by n, the shell will accepts any 3 characters before continuing.

Controlling the visibility of entered text

This is useful when we are entering sensitive data such as PIN (or) password.

for this we can use silent option -s.

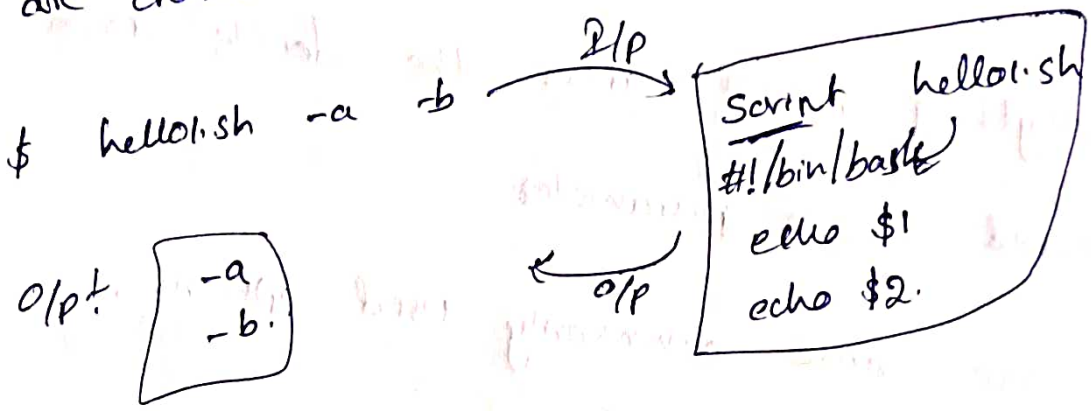
Ex: read -sn1 -p "press any key to exit".

Even if we enter character we cannot see that.

Passing options

what are options? How are they different from parameters?

options are charaters with a single dash (-) before them.



options can be passed with a value, like this

$ hellosh -a -b 20 -C → value of -b is 20

If we directly execute it like before, then we will get following o/p which is not as expected.

-a
-b
20
-c

→ Here, there is a separate line for 20 which we didn't expected. Normally the value should be stored inside it.

so, inorder to achieve this we have other concept as following.

Passing parameters with options

To pass parameters along with options simultaneously, we must separate them with a double dash.

```
$ kellorsh -a -b -c -- P1 P2 P3
```

So, Anything passed after the double dash is treated as a parameter.

There are some commonly used options.

1. -a : List all items
2. -c : Get a count of all items
3. -d : output directory
4. -e : Expand items
5. -f : specify a file
6. -h : Show the help page
7. -i : Ignore the character case
8. -l : list a text
9. -o : send output to a file
10. -q : keep silent; don't ask the user
11. -r : process something recorsively
12. -s : use stealth mode
13. -v : use verbose mode
14. -x : specify an executable
15. -y : Accept without prompting me.

Simple decision paths using command-line lists.

command-line lists are 2 or more statements that are joined using either AND or OR notations!

- && : AND
- || : OR

Ex: $ test $PWD == $HOME || cd $HOME.

It ensures whether the corre

test command

The test command will always return either True (or) False (or) 0/1 respectively.

* The basic Syntax is
  test EXPRESSION
* we can invert test command
  test ! EXPRESSION

* when the test command is run without any expressions to evaluate, then the test will return false. so the exit status will be 1, even though no error o/p is shown.

→ If we need to include multiple expressions, this can be done using AND or OR together, using the -a and -o options.

test EXPRESSION -a EXPRESSION

test EXPRESSION -o EXPRESSION.

we can also write this as a shorthand version, replacing the test with square brackets to surround the expression.

[EXPRESSION]

TESTING STRINGS

we can test for the quality or inequality of two strings.

Ex: to test the root user

test $USER = root

we can write it as

[$USER = root]

Note: we must put a space b/w each bracket and the inner testing condition as previously shown.

Testing Integers

Another way of testing the I/p of a script is to count the numbers of positional parameters and also test if the number is above 0.

test $# -gt 0 ⟶ [$# -gt 0]

when in a relationship, the top positional parameters of the $# variable represent the no of parameters passed to the script.

There are many tests that can be done for numbers

* number1 -eq number2 : checks if number1 equal to number2.
* number1 -ge number2 : checks if number1 is greater than or equal to number2
* number1 -gt number2 : " " greater than "
* number1 -le number2 : " " less than or equal to "
* number1 -lt number2 : " " less than " "
* number1 -ne number2 : " " not equal " "

Testing file types.

while testing for values, we can test for the existence of a file or a file type.

* -h : tests that the file has a link.
* -d : this shows that it's a directory
* -e : this shows that the file exists in any form
* -x : this shows that the file is executable
* -f : this shows that the file is a regular file.

* -r : This shows that the file is readable.
* -p : This shows that the file is a name pipe
* -b : This shows that the file is a block device.
* file1 -nt file2 : This checks if file1 is newer than file2
* file1 -ot file2 : This checks if file1 is older than file2
* -O file : This checks if the logged-in user is the owner of the file
* -c : This shows that the file is a character device.

Conditional Statements using if:

The code within the if statement will run only when the condition evaluates to True, and the end of the if block is denoted with fi (backward of if).

Syntax:

```
if condition ; then
    Statement 1
    Statement 2
fi
```

Indentation is not mandatory, but it helps readability.
semicolon is mandatory if we write then in same line of if statement, otherwise no need.

Extending if with else.

Syntax:

```
if condition ; then
    statement
else
    statement
fi
```

test command with if command.

checking strings.

* if [$string1 = $string2] : checks if str1 is identical to str2
* if [$string1 != $string2] : checks if str1 is not identical to str2
* if [$string1 \< $string2] : " " less than "
* if [$string1 \> $string2] : " " greater than "

Note: less than & greater than should be escaped with a backslash (\) as if it shows a warning.

* if [-n $string1] : checks if string1 is longer than zero
* if [-z $string1] : checks if string1 has zero length.

Ex:
```
#!/bin/bash
if [ "mokhtar" = "Mokhtar" ]
then
echo "Strings are identical"
else
echo "string are not identical"
fi
```

* To check the string length

```
#!/bin/bash
if [ -n "gani" ]
then
echo "string length is greater than zero"
else
echo "string is zero length"
fi
```

* To check for a length of zero, you can use the -z test

```
#!/bin/bash
if [ -z "gani" ]
then
echo "string length is zero"
else
echo "string length is not zero"
fi
```

elif

```
if condition; then
    statement
elif condition; then
    statement
else
    statement
fi
exit 0
```

Using case statements:

Rather than using multiple elif statements, a case statement may provide a simpler mechanism when evaluations are made on single expression.

Syntax: case expression in

```
    case 1)
        statement 1
        statement 2
    ;;
    case 2)
        statement 1
        statement 2
    ;;
    *)
        statement 1
    ;;
esac
```

car.sh

Ex:

```
#! /bin/bash
car = $1 → positional parameters, it takes the first argument from the passed arguments.

case $car in
"BMW")
    echo "this is BMW" ;; → This is used to break the statement
"Mercedese")
    echo "this is mercedese" ;;
*)
    echo "this is not a car (or) not a car listed" ;;
esac → end of the case statement.
```

Start of case statement

default statement

Execution

```
$ ./car.sh BMW
this is BMW

$ ./car.sh tesla
this is not a car (or) not a car listed.
```

while loops

Syntax:

```
while [condition]
do
    command 1
    command 2
    command 3
done
```

Ex:

```
n=1
while [ $n -le 10]
do
    echo "$n"
    n=$(( n+1 ))
done
```

done → end of while loop.

Increment (or) Decrement in Bash

In bash there are multiple ways to increment / decrement a variable.

Increment

```
i= $(( i+1 ))
(( i=i+1 ))   (or)  (( i+=1 ))  (or)  ((i++))
let "i=i+1"   (or)  let "i+=1"
```

↓ reassign the variable.

↓ inplace operation

Decrement

```
i= $(( i-1 ))
(( i=i-1 ))   (or)  (( i-=1 ))  (or)  ((i--))
let "i=i-1"   (or)  let "i-=1"
```

↓ reassign the variable.

↓ inplace operation

Bash Arrays.

Array is a variable that can store multiple variables.

* Numerically we can retrieve last element of array by using index of -1.

* unlike most of the programming languages, Bash array elements don't have to be of the same data type.

* Bash does not support multidimensional arrays and also we can't have array elements that are also arrays.

Creating Bash Arrays.

Arrays in bash can be initialized in different ways

① To explicitly declare an array, use the declare builtin

```
declare -a array_name

array_name[index_1] = value_1
array_name[index_2] = value_2
array_name[index_n] = value_n
```

② array_name =(element_1 ele2 ele3 element_N)

Note: there are no comma's b/w elements.

Associative arrays.

Associative arrays are referenced using arbitrary strings, whereas indexed arrays are referenced using integers.

syntax:

```
declare -A array_name

array_name [fruit] = mango
array_name [bird] = parrot
array_name [flower] = Rose
```

Accessing in o/p terminal:

```
$ echo ${array_name[fruit]}.
mango.
```

Array Operations.

An element can be referenced using the following syntax

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

The curly braces ${} are required to avoid shells filename expansion operators.

* If we use @ or * as an index, then the word expands to all members of array.

echo "${my_array[@]}" → prints all elements.

* Difference b/w $@, $*, "$@", "$*"

$* is a single string

$@ is an actual array.

Ex: $ test.sh one two "three four" → parameter

script

```
for a in "$*"; do
    echo $a;
done
```
→ this treats the parameters as one long quoted string
→ o/p: one two three four

```
for a in $*; do
    echo $a;
done
```
→ It treats each element as unquoted string
→ o/p: one
two
three
four

```
for a in "$@"; do
    echo $a;
done
```
→ It treats each element as quoted string.
o/p: one
two
three four

```
for a in $@; do
    echo $a;
done
```
→ It treats each element as unquoted string
one
two
three
four.

Array indices

To print the list of array indices,

```
${!array_name[@]}
```

Array Length.

To get array length, use following syntax.

```
${#array_name[@]}
```

Loop through array

```
declare -a my_array=("Mango" "Banana" "orange")

for i in "${my_array[@]}"
do
    echo "$i"
done
```

→ for values.

o/p: Mango
Banana
orange.

```
for i in "${!my_array[@]}"
do
    echo "$i"
done
```

→ for indices.

Another way for looping (C style).

```
declare -a my_array = ("mango" "Banana")

length = ${#my_array[@]}

for ((i=0; i< ${length}; i++ ))

do
    echo $i ${my_array[$i]}

done.
```

Adding Element

```
my_array[index_n] = "new element"
```

Deleting element

```
unset my_array[index]
```

For loop

Loop over strings.

```
for element in Hydrogen Helium Nitrogen

do
    echo "Element: $element"

done.
```

loop over number range:

includes start → end. also includes ⟶ {start .... END}

```
for i in {0...3}
do
   echo "$i"
done
```

o/p :
0
1
2
3

{start ... END... Increment}

```
for i in {0...20...5}
do
   echo "$i"
done
```

o/p :
0
5
10
15
20

break and continue statements.

* the break statement terminates the current loop & passes program control to the statement that follows the terminated statement.

```
for ele in Hydrogen Helium Lithium
    if [[ "$ele" == 'Lithium' ]]; then
        break
    fi
    echo "Element : $ele"
done
echo "All Done!"
```

o/p =

Element : Hydrogen
Element : Helium
All done!

* The continue statement exits the current iteration of a loop and passes program control to the next iteration.

```
for i in {1..5}; do
    if [[ "$i" == '2' ]]; then
        continue
    fi
    echo "number: $i"
done
```

o/p

```
number: 1
number: 3
number: 4
number: 5
```

Bash functions

Functions in bash are declared in two formats.

①
```
function_name ( ) {
    commands
}
```

single line version!

```
function_name ( ) { commands; }
```

②
```
function function_name {
    commands
}
```

single line version!

```
function function_name { commands; }
```

Ex: #!/bin/bash → function definition

```
hello_world(){
    echo 'hello, world'
}
```

hello_world → function call

Variable Scope

In bash, all variables by default are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the local keyword and can be used only inside that function. You can have local variables with the same name in different functions.

Ex:

```
#!/bin/bash
var1='A'
var2='B'
my_function(){
    local var1='c'
    var2='b'
    echo "Inside function: var1: $var1, var2: $var2"
}
echo "Before executing function: var1: $var1, var2: $var2"
my_function
echo "After executing function: var1: $var1, var2: $var2"
```

O/P

```
Before executing fn: var1: A, var2: B
Inside fn: var1: c, var2: b
After fn: var1: A, var2: b
```