

System Commands
Professor. Gandham Phanikumar
Metallurgical and Materials Engineering
Indian Institute Technology, Madras
Bash Scripts Part 2A

(Refer Slide Time: 0:14)

Shell programming



More features in bash scripts

o



Welcome to the second session on shell programming. In this session, we will look at some more features of bash scripts. And you can try them out, watching what I am doing and then copying those codes, modifying them, and adapting them to your needs. So, let us get started with the features in bash scripts that go beyond what we have covered in the previous session.

(Refer Slide Time: 0:41)

Debugging



```
set -x  
./myscript.sh
```

Prints the command before
executing it

Place the **set -x** inside
the script

```
bash -x ./myscript.sh
```

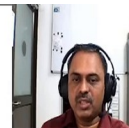
○



There is nothing really called as debugging a shell script, what is of course possible is to set the minus x option either at the bash prompt or inside the script, so that every command that is executed is first printed out and then the output is shown. So, in situations where there is a loop and multiple conditions, then this option could help you in identifying if there was any behavior that you were not expecting.

(Refer Slide Time: 1:13)

Combining conditions



```
[ $a -gt 3 ] && [ $a -gt 7 ]
```

```
[ $a -gt 3 ] || [ $a -gt 7 ]
```

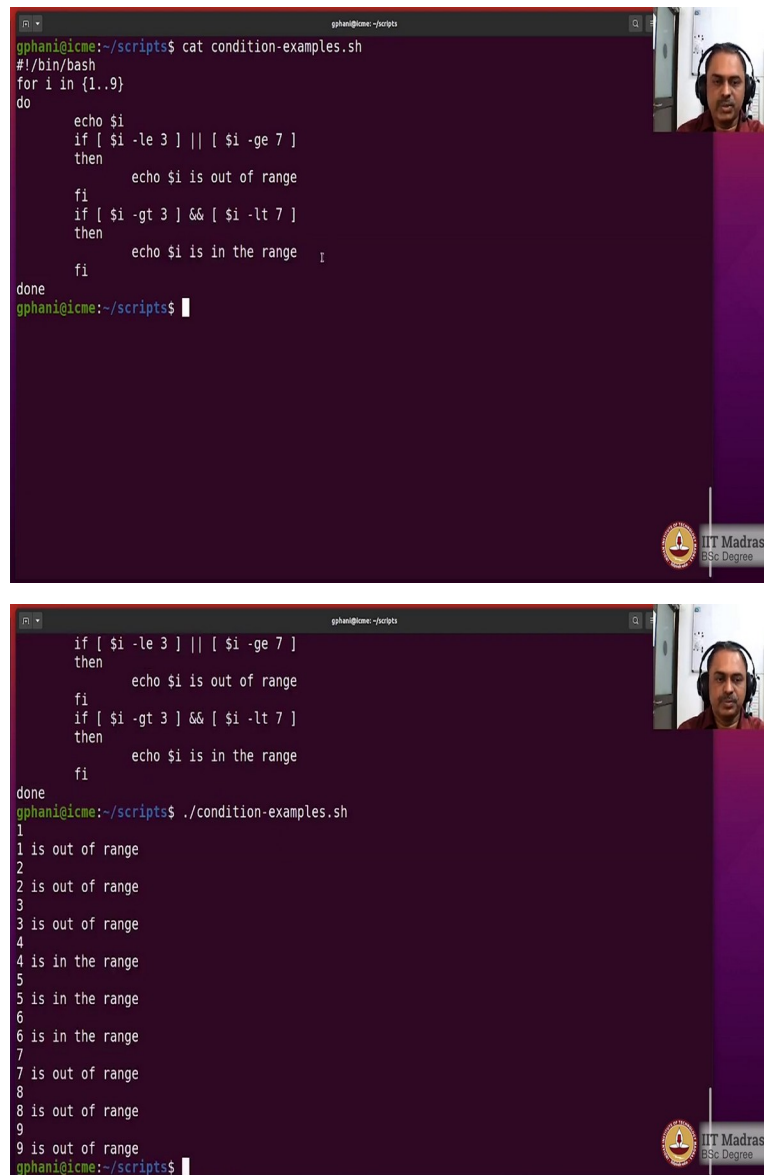
○



You can combine multiple test conditions for the loops by using the double ampersand or the double pipe symbol. And these behaves similar to what we have discussed earlier, namely, that

the double ampersand would work as the logical AND operator the double pipe would work as the logical OR operator. So, the cost condition that is shown here would work for the values of the variable A, such as 4, 5 and 6. Let us try this out on the shell prompt using a example script and see.

(Refer Slide Time: 1:51)



```
gphani@cme:~/scripts$ cat condition-examples.sh
#!/bin/bash
for i in {1..9}
do
    echo $i
    if [ $i -le 3 ] || [ $i -ge 7 ]
    then
        echo $i is out of range
    fi
    if [ $i -gt 3 ] && [ $i -lt 7 ]
    then
        echo $i is in the range
    fi
done
gphani@cme:~/scripts$
```

```
if [ $i -le 3 ] || [ $i -ge 7 ]
then
    echo $i is out of range
fi
if [ $i -gt 3 ] && [ $i -lt 7 ]
then
    echo $i is in the range
fi
done
gphani@cme:~/scripts$ ./condition-examples.sh
1
1 is out of range
2
2 is out of range
3
3 is out of range
4
4 is in the range
5
5 is in the range
6
6 is in the range
7
7 is out of range
8
8 is out of range
9
9 is out of range
gphani@cme:~/scripts$
```

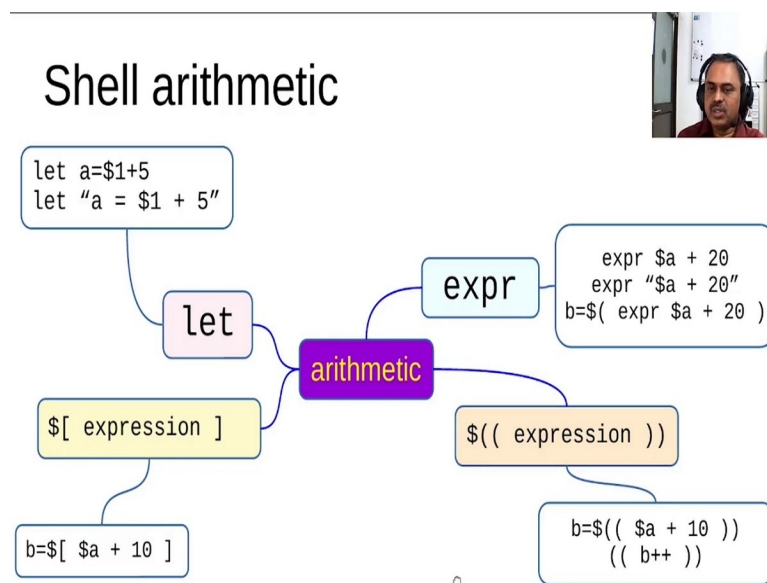
The script that we would be using to practice the combination of test conditions is shown here. So, we are looping through the values of i from 1 to 9 and then we are printing that value. After that, we are checking for the value of i either it is less than 3 less than or equal to 3 or it is greater

than or equal to 7 in that case, it says i is out of range. And if it happens to be greater than 3 less than 7, then i is in the range.

So, when we execute that, you can see the behavior as expected 1 2 3 are out of the range, because less than or equal to then 4 5 6 are within the range and again 7 8 9 out of range. So, you can see that one can combine the conditions in this manner by using the double pipe of our OR double ampersand for AND. And you can also see that we are doing nesting of loops, which is quite straightforward. Any set of commands can as well be other loops.

And ensure that you have close to the loops properly. So, that there is no confusion later on when you want to read your own code. So, indenting always helps in identifying which loop is ending where in your basket.

(Refer Slide Time: 3:16)



Though the shell environment is originally meant for running system commands, working with files, network operations and so on, sometimes we also need to do some arithmetic inside the shell. And there are plenty of such possibilities. Here on the screen you are seeing four different such possibilities. One is using the let command, where you could give the expression performing an arithmetic operation either without quotes or with quotes.

If you are doing it without quotes, then spaces should not be inserted on either sides of the equal to sign because that is same as assigning value to a variable at the shell prompt. And in case you are using quotes, then you can give spaces on either sides of the equal to sign to make your

expression more legible. There is also a construction which goes with the square brackets after the dollar. And you can write an expression inside that and an example is here, where dollar a plus 10 is evaluated. And the output is then assigned to the variable b.

expr is yet another command like the let command, which would actually perform the arithmetic operation and print the output like any other command, which can be executed and the output can be assigned to a variable. The command substitution operation available with dollar followed by parenthesis can be used to also with expr.

Very often expr is used on the shell to just display the output of the operation but you can also assign it to a variable. And you can also use the double parenthesis notation where you can give arithmetic operation within the double parenthesis. And there is a rich set of operators available for this construction. If you give this expression without the dollar, then it means that you are not intending to return the value to be assigned to another variable.

So, operators such as the plus plus or minus minus could be used to increment or decrement a variable very elegantly by using the double parenthesis operator. Now, you can see that there is a very rich set of arithmetic operations that are possible. So, let us go ahead and see what are the various operations available using the expr command because that is the most widely used command in the bash scripts to perform arithmetic operations.

(Refer Slide Time: 5:55)

expr command operators - 1	
a + b	Return arithmetic sum of a and b
a - b	Return arithmetic difference of a and b
a * b	Return arithmetic product of a and b
a / b	Return arithmetic quotient of a divided by b
a % b	Return arithmetic remainder of a divided by b
a > b	Return 1 if a greater than b; else return 0
a >= b	Return 1 if a greater than or equal to b; else return 0
a < b	Return 1 if a less than b; else return 0
a <= b	Return 1 if a less than or equal to b; else return 0
a = b	Return 1 if a equals b; else return 0

So, you could use the plus minus star and forward slash to perform the sum difference product and quotient operations. And remember, these are all only for integer operations. If you wanted floating point operations, you must use the bench calculator, which we would also look at in this exercise, the percent operator is meant to take the remainder b greater than or greater than equal to or less than or less than or equal to or equal to or not equal to these are the operators which are actually not going to look at the actual values of a and b.

But to perform a logical comparison and return a value of 1 or 0, depending upon the particular operation. For example, a greater than b would return 1 if a is actually larger in magnitude than b otherwise it will return 0, a greater than or equal to b would return 1 if the numerical value of a is greater than or equal to b, otherwise it would return 0 so, on for all other operators below.

(Refer Slide Time: 6:59)

expr command operators - 2	
a b	Return a if neither argument is null or 0; else return b
a & b	Return a if neither argument is null or 0; else return 0
a != b	Return 1 if a is not equal to b; else return 0
str : reg	Return the position upto anchored pattern match with BRE str
match str reg	Return the pattern match if reg matches pattern in str
substr str n m	Return the substring m chars in length starting at position n
index str chars	Return position in str where any one of chars is found; else return 0
length str	Return numeric length of string str
+ token	Interpret token as string even if its a keyword
(exprn)	Return the value of expression exprn

You also have a pipe operator, where if neither argument is null or 0, then a will be returned, otherwise b will be returned. The ampersand operator is going to return a if neither argument is null or 0, otherwise, it will return actually 0 not b. So, you can see that there is a subtle difference between the pipe and ampersand operators. And then there are some operations available with strings.

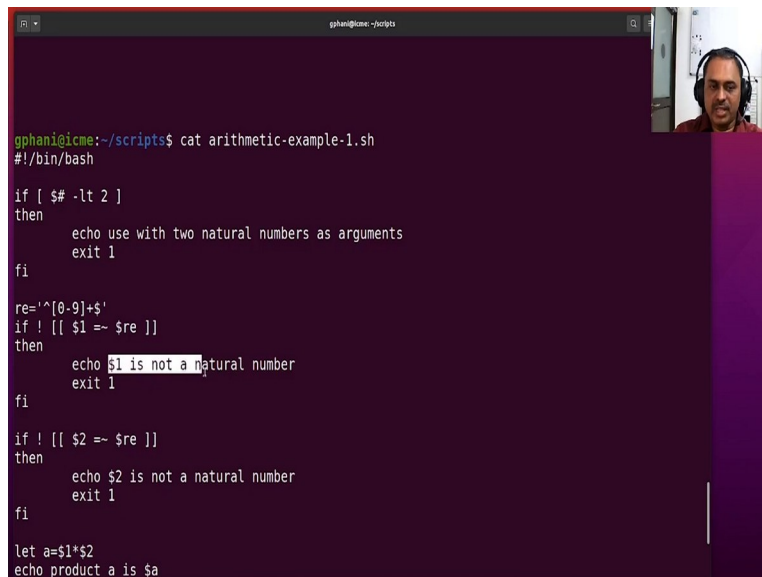
So, string colon regular expression can be used using the basic regular expression engine that is very limited set of regular expressions and the position up to which such a matching is possible would then be returned. You can use a command match string and then the regular expression to

check if the regular expression given is going to match the string. And if so, then whatever is matched is then returned.

Substring followed by string followed by 2 integers would return the substring from the string starting from the position m and as long as n characters. So basically, it is like taking a slice of the string by mentioning the position as well as the length of the substring. You could also use the index command to check what is the first occurrence of any of the characters listed in the third argument which are available in the string, which is displayed in the middle.

You could also find out the length of a string using the expression. So, `expr length string` would give you the length of the string. And there are also two special commands available to override the default behavior of keywords that are available in the bash environment. So, let us now look at some of these examples. Both the arithmetic as well as `expr` commands.

(Refer Slide Time: 9:02)



```
gphani@icme: ~/scripts
gphani@icme:~/scripts$ cat arithmetic-example-1.sh
#!/bin/bash

if [ $# -lt 2 ]
then
    echo use with two natural numbers as arguments
    exit 1
fi

re='^[0-9]+$'
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
```

```
gphanigme:~/scripts
then
    echo use with two natural numbers as arguments
    exit 1
fi

re='^[0-9]+$'
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
(( a++ ))
echo product a incremented is $a

let b=$1**$2
echo power is $b

c=$(( $1 + $2 + 10 ))
echo sum+10 is $c
```

```
gphanigme:~/scripts$ cat arithmetic-example-1.sh
#!/bin/bash

if [ $# -lt 2 ]
then
    echo use with two natural numbers as arguments
    exit 1
fi

re='^[0-9]+$'
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
(( a++ ))
echo product a incremented is $a
```



```
gphani@cme: ~/scripts
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
(( a++ ))
echo product a incremented is $a

let b=$1**$2
echo power is $b

c=$(( $1 + $2 + 10 ))
echo sum+10 is $c

d=$((expr $1 + $2 + 20))
echo sum+20 is $d

f=$(( $1 * $2 * 2 ))
echo product times 2 is $f
gphani@cme:~/scripts$ ./arithmetic-example-1.sh
use with two natural numbers as arguments
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 12
use with two natural numbers as arguments
gphani@cme:~/scripts$
```

```
gphani@cme: ~/scripts
let b=$1**$2
echo power is $b

c=$(( $1 + $2 + 10 ))
echo sum+10 is $c

d=$((expr $1 + $2 + 20))
echo sum+20 is $d

f=$(( $1 * $2 * 2 ))
echo product times 2 is $f
gphani@cme:~/scripts$ ./arithmetic-example-1.sh
use with two natural numbers as arguments
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 12
use with two natural numbers as arguments
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 12 4
product a is 48
product a incremented is 49
power is 20736
sum+10 is 26
sum+20 is 36
product times 2 is 96
gphani@cme:~/scripts$ ./arithmetic-example-1.sh asdf4
use with two natural numbers as arguments
gphani@cme:~/scripts$ ./arithmetic-example-1.sh asdf 4
asdf is not a natural number
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 4 asdf
asdf is not a natural number
gphani@cme:~/scripts$
```

So, let us look at an arithmetic example here. So we have a script arithmetic example, where I am checking for the number of arguments. And if it is less than 2, then I am giving an error saying that it should be used, the script should be used with 2 natural numbers as arguments. And then I have a regular expression here, which looks for the starting of the string should be with a digit, and then any number of such digits and then the string should be ending.

That means that there should be nothing other than digits in the string, which means it should be an integer. And what I am doing here in this command is this part is going to check whether the first argument is going to match the regular expression. The asterisk in the front is a negation,

which means that I am asking in case the first argument does not match the regular expression, then we are giving an output saying that first argument is not a natural number.

And we are repeating the same thing for the second argument also. So what I am doing is basically I am checking whether the first and second arguments are natural numbers or not. And if they are not, then I am exiting, so that we are not proceeding further with the script. And if they are natural numbers, then what we are doing here is using the `let` command, we are finding the product of the two numbers and printing that product out.

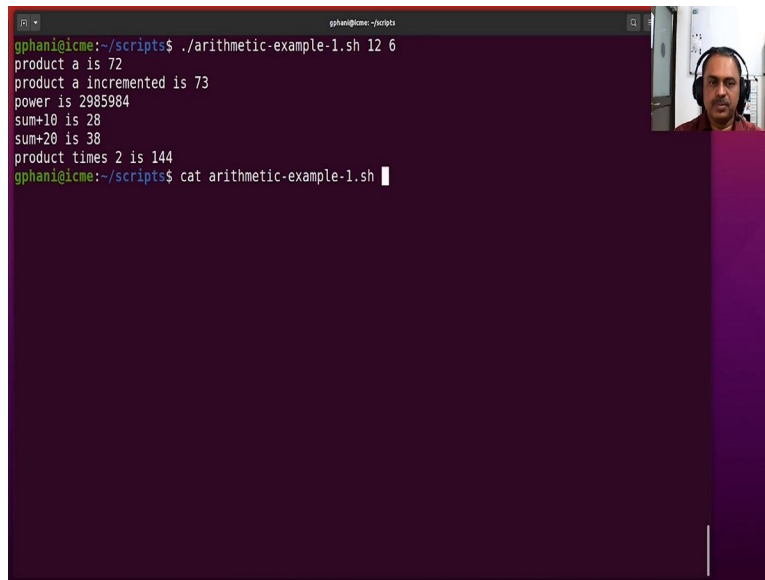
After that, we are actually incrementing the product by one number using the plus plus operator using the double parenthesis option and then displaying that value also below. And then we also have the one more operation where we are using the power function. So, the first argument raised to the power of second argument and that is also displayed to show the possibilities of operators using the `let` command.

After that the square bracket as a feature for the arithmetic operations is shown here. We take the sum of the two arguments and add 10 to it. And then we also use the `expr` command here to show the value of the two arguments ended and 20. And then we also have last way of showing the mathematics namely by using a double namely by using the double parenthesis operators. So, what we will do is that we will execute this.

When executed without any options, the output is used with two natural numbers as arguments. So, here you see that I have not given any argument. So it should come out by saying that I should use two natural numbers. So, I will give only one natural number. So, I will say 12. That is all and it still gives that option because I have given only one argument. So, now I will do one thing I will actually give two arguments. I will give 4 as a second argument and then it would go on to run.

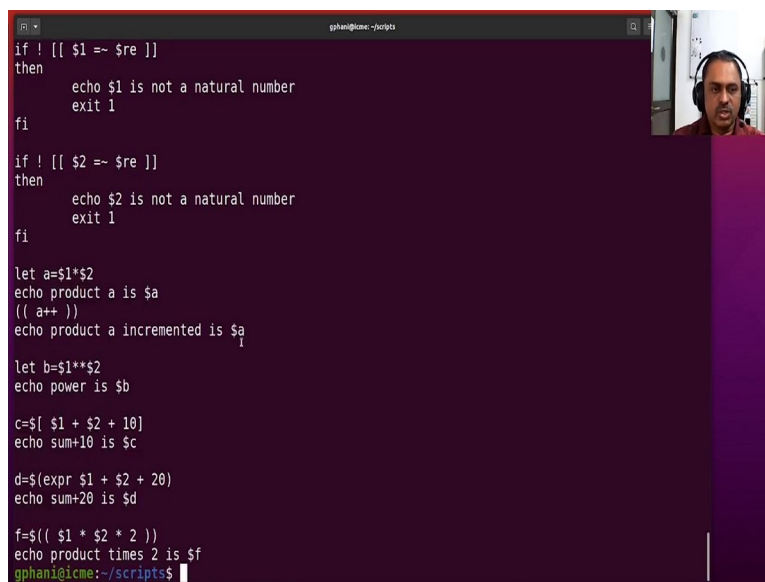
Now, this regular expression check is now very nice what you can do is for example `asdf` and then 4. You can see that the first argument is `asdf` is not a natural number. The second argument for is accepted. So, let us say I swap these 2 and you can see that again it says that the second argument is not acceptable first is accepted. So, it is already checking for the natural numbers and then only proceeding further.

(Refer Slide Time: 12:31)



A terminal window titled 'gphani@icme: ~/scripts' showing the execution of a script named 'arithmetic-example-1.sh' with arguments '12' and '6'. The script outputs several arithmetic results. A small video inset in the top right corner shows a man with a beard and headphones.

```
gphani@icme:~/scripts$ ./arithmetic-example-1.sh 12 6
product a is 72
product a incremented is 73
power is 2985984
sum+10 is 28
sum+20 is 38
product times 2 is 144
gphani@icme:~/scripts$ cat arithmetic-example-1.sh
```



The same terminal window showing the source code of 'arithmetic-example-1.sh' as viewed with 'cat'. The code includes conditional checks for natural numbers, variable assignments, and arithmetic calculations. The same video inset is present.

```
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
(( a++ ))
echo product a incremented is $a

let b=$1**$2
echo power is $b

c=$(( $1 + $2 + 10 ))
echo sum+10 is $c

d=$((expr $1 + $2 + 20))
echo sum+20 is $d

f=$(( $1 * $2 * 2 ))
echo product times 2 is $f
gphani@icme:~/scripts$
```

```
gphani@cme: ~/scripts
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
(( a++ ))
echo product a incremented is $a

let b=$1**$2
echo power is $b

c=$(( $1 + $2 + 10 ))
echo sum+10 is $c

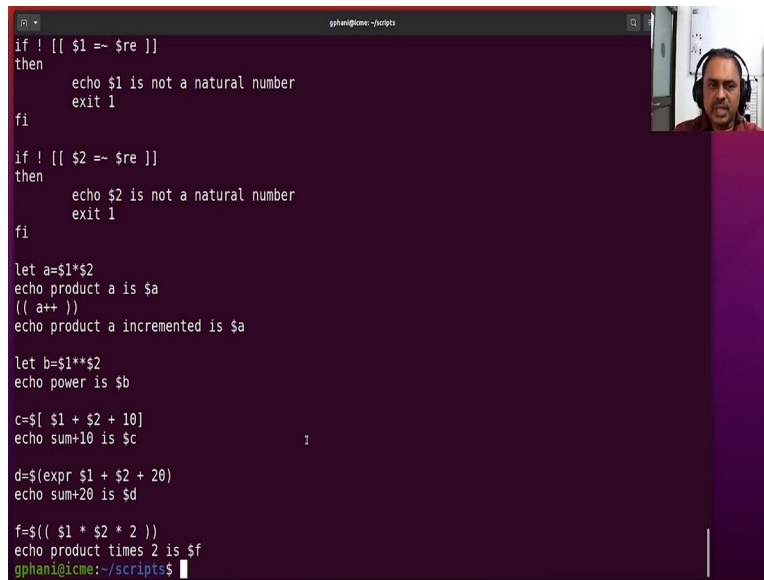
d=$((expr $1 + $2 + 20))
echo sum+20 is $d

f=$(( $1 * $2 * 2 ))
echo product times 2 is $f
gphani@cme:~/scripts$
```

```
gphani@cme:~/scripts$ ./arithmetic-example-1.sh asdf4
use with two natural numbers as arguments
gphani@cme:~/scripts$ ./arithmetic-example-1.sh asdf 4
asdf is not a natural number
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 4 asdf
asdf is not a natural number
gphani@cme:~/scripts$
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 12 6
product a is 72
product a incremented is 73
power is 2985984
sum+10 is 28
sum+20 is 38
product times 2 is 144
gphani@cme:~/scripts$ cat arithmetic-example-1.sh
#!/bin/bash

if [ $# -lt 2 ]
then
    echo use with two natural numbers as arguments
    exit 1
fi

re='^[0-9]+$'
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi
```



```
gphanigme:~/scripts
if ! [[ $1 =~ $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 =~ $re ]]
then
    echo $2 is not a natural number
    exit 1
fi

let a=$1*$2
echo product a is $a
(( a++ ))
echo product a incremented is $a

let b=$1**$2
echo power is $b

c=$(( $1 + $2 + 10 ))
echo sum+10 is $c

d=$((expr $1 + $2 + 20))
echo sum+20 is $d

f=$(( $1 * $2 * 2 ))
echo product times 2 is $f
gphanigme:~/scripts$
```

So, let us go ahead and let it actually work with the arguments, which are natural numbers. And here I will actually also do the so the very first operation is to take the product and echo that so you see that the product has come to 12 6 are 72. Then the second operator here is to increment it by 1 and then we are printing it saying that product a incremented is dollar a. So that is what is saying here product a increment it is 73; 72 plus 1.

And then we have the power operator here b so 12 raised to the power of 6 and that is coming out to be here. And the sum 12 plus 6 is being shown here plus 10. So, this is showing you 12 plus 6 plus 10 and you can see that is 28 here. And the last arithmetic operation is here, the next operator is to add 20 to the sum. So, that is coming up here some plus 20 is 38 the last arithmetic operation is to take the product of the two numbers and make it twice.

So, you will see that 12 into 6; 72 into 144. So, you can see that you can perform arithmetic operations using very different syntax of possibilities that are available and you can choose whichever is convenient to you and go on to explore numerics within the bash environment remember that this is only for the integers.

(Refer Slide Time: 14:22)

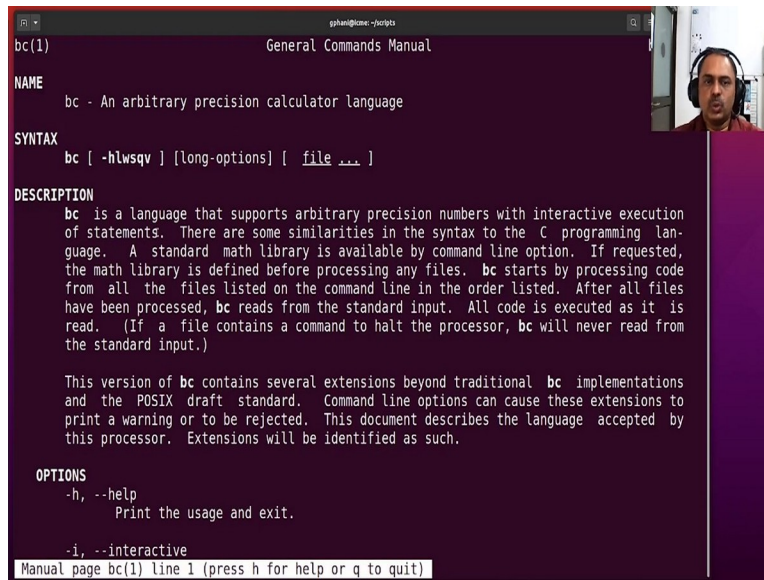
```
gphani@cme:~/scripts$ bc -l
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundat:
.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
12^6
2985984
12.5^6
3814697.265625
12.6/3.2
3.937500000000000000
^C
(Interrupt) use quit to exit.
quit
gphani@cme:~/scripts$
```

```
asdf is not a natural number
gphani@cme:~/scripts$
gphani@cme:~/scripts$ ./arithmetic-example-1.sh 12 6
product a is 72
product a incremented is 73
power is 2985984
sum+10 is 28
sum+20 is 38
product times 2 is 144
gphani@cme:~/scripts$ cat arithmetic-example-1.sh
#!/bin/bash

if [ $# -lt 2 ]
then
    echo use with two natural numbers as arguments
    exit 1
fi

re='^[0-9]+$'
if ! [[ $1 == $re ]]
then
    echo $1 is not a natural number
    exit 1
fi

if ! [[ $2 == $re ]]
then
    echo $2 is not a natural number
    exit 1
```



```
bc(1)                                General Commands Manual

NAME
  bc - An arbitrary precision calculator language

SYNTAX
  bc [ -hlwsqv ] [long-options] [ file ... ]

DESCRIPTION
  bc is a language that supports arbitrary precision numbers with interactive
  execution of statements. There are some similarities in the syntax to the C
  programming language. A standard math library is available by command line
  option. If requested, the math library is defined before processing any files.
  bc starts by processing code from all the files listed on the command line in
  the order listed. After all files have been processed, bc reads from the
  standard input. All code is executed as it is read. (If a file contains a
  command to halt the processor, bc will never read from the standard input.)

  This version of bc contains several extensions beyond traditional bc
  implementations and the POSIX draft standard. Command line options can cause
  these extensions to print a warning or to be rejected. This document describes
  the language accepted by this processor. Extensions will be identified as
  such.

OPTIONS
  -h, --help          Print the usage and exit.

  -i, --interactive

Manual page bc(1) line 1 (press h for help or q to quit)
```

Now, there is a bench calculator available you can load it with the minus l option to load the math library. And you can perform some operations here so 12 arrays per of 6 you can see that it is giving you a number 298 5984 which is actually matching what is here. So, you could also do these things with a floating point number. So, that would work with floating point operations also.

So, if you want to do floating point operations are explanations of real numbers. So, then you could also use the bench calculator. So, look at the man bc. So, it is an arbitrary position calculator very powerful. So, we can also use this within the shell scripts, if we want to do those kind of arithmetic's.

(Refer Slide Time: 15:11)

```
gphani@icme: ~/scripts
echo $ans
ans=$( expr $a = $b )
echo $ans

ans=$( expr $a != $b )
echo $ans

ans=$( expr $a \| $b )
echo $ans

ans=$( expr $a \& $b )
echo $ans

str="octavio version as in Jan 2022 is 6.4.0"
reg="[o0]ctav[aeiou]*"
ans=$( expr "$str" : $reg )
echo $ans

ans=$( expr substr "$str" 1 6 )
echo $ans

ans=$( expr index "$str" "vw" )
echo $ans

ans=$( expr length "$str" )
echo $ans
gphani@icme:~/scripts$
```

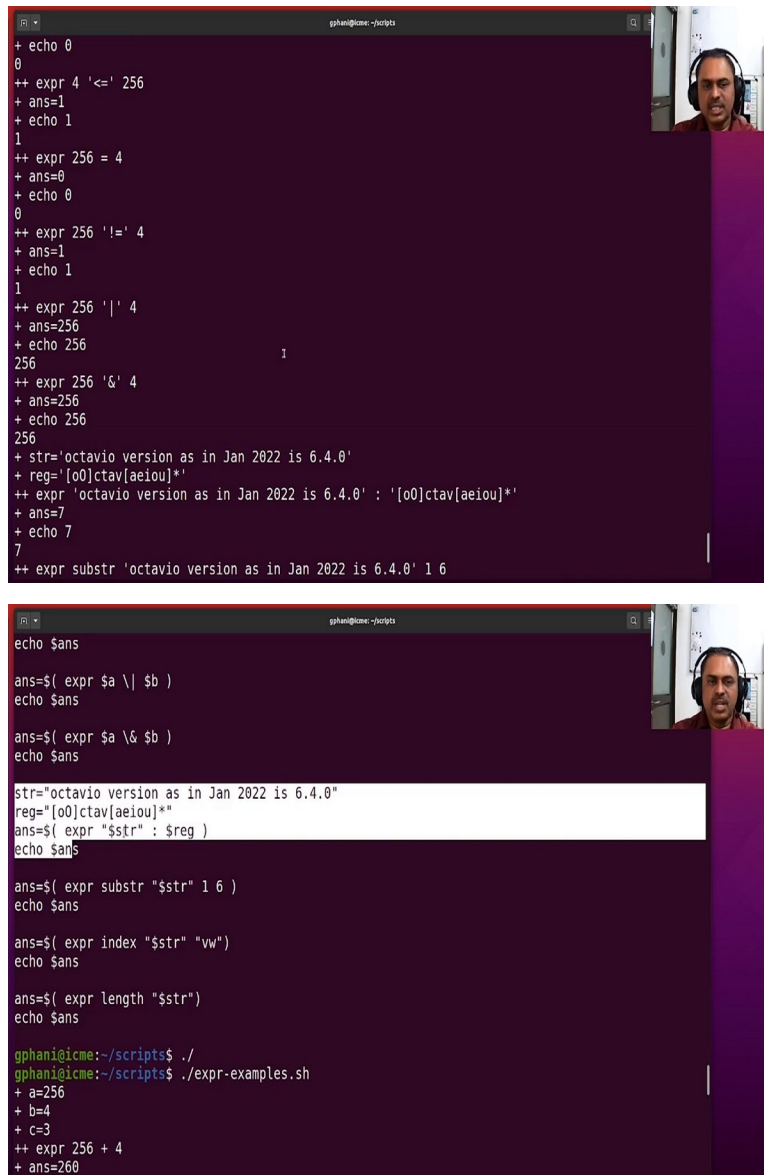
```
gphani@icme:~/scripts$ cat expr-examples.sh
#!/bin/bash
set -x
a=256
b=4
c=3

ans=$( expr $a + $b )
echo $ans

ans=$( expr $a - $b )
echo $ans

ans=$( expr $a \* $b )
echo $ans

ans=$( expr $a / $b )
echo $ans
```

```
+ echo 0
0
++ expr 4 '<=' 256
+ ans=1
+ echo 1
1
++ expr 256 = 4
+ ans=0
+ echo 0
0
++ expr 256 '!=' 4
+ ans=1
+ echo 1
1
++ expr 256 '|' 4
+ ans=256
+ echo 256
256
++ expr 256 '&' 4
+ ans=256
+ echo 256
256
+ str='octavio version as in Jan 2022 is 6.4.0'
+ reg='[o0]ctav[aeiou]*'
++ expr 'octavio version as in Jan 2022 is 6.4.0' : '[o0]ctav[aeiou]*'
+ ans=7
+ echo 7
7
++ expr substr 'octavio version as in Jan 2022 is 6.4.0' 1 6

echo $ans

ans=$( expr $a \| $b )
echo $ans

ans=$( expr $a \& $b )
echo $ans

str="octavio version as in Jan 2022 is 6.4.0"
reg="[o0]ctav[aeiou]*"
ans=$( expr "$str" : $reg )
echo $ans

ans=$( expr substr "$str" 1 6 )
echo $ans

ans=$( expr index "$str" "vw" )
echo $ans

ans=$( expr length "$str" )
echo $ans

gphani@icme:~/scripts$ ./
gphani@icme:~/scripts$ ./expr-examples.sh
+ a=256
+ b=4
+ c=3
++ expr 256 + 4
+ ans=260
```

Now, let us explore the expr functionality. So, in the expr functionality, I have got a long list of commands, where we are testing out every single operation, but I am also telling you how to actually use that without confusing the shell by using the characters which are special to the shell. So, we are actually showing all these things and here I have put a set minus x option so that you can actually run it a bit the debugging namely to just display the command before you run it.

So, you can see that it has gone very long. So, every time a command is run, there is a character plus that is kept in front and the output alone will be without the plus in the front. So, I have done 3 commands here assigning values of a and b and c, and then I am running expr 256 plus 4.

And you will see that it gives you the answer here. And 256 minus 4 the answer is here 256 multiplied by 4 answer is here 256 divided by 4 answer is here.

Now, you can see that from the script, the way we have given the multiplication is with a backslash to escape that. So, that let us remember that and then you can see that when we come to operations which involve greater than symbol, or greater than or equal to the answers are actually 1 or 0. So, you can say if you are asking 256 greater than 4, is it true? Yes. So answer is 1, 4 is greater than or equal 256, answer is false. So answer is 0.

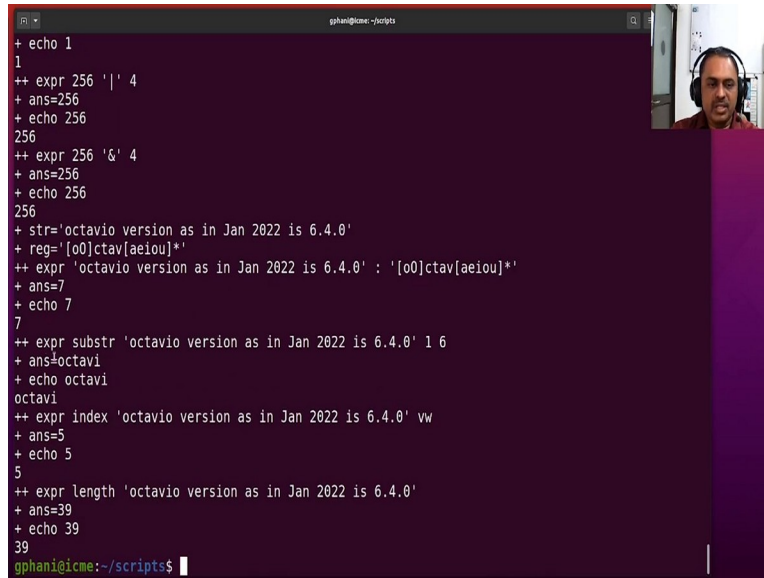
Similarly, you are asking, is it less 256 less than 4, and answer is 0. Is 4 less than or equal 256? Answer is yes. And 1. And is 256 equal to 4 answer is no 0, and not equal to answer is yes 1. And the pipe symbol. If the both arguments are available, then the first argument is returned. So, you see that 256 is the answer. That is the first argument. An ampersand symbol is when also you have got the similar kind of behavior. Only when you look at the second argument, then you have a difference there, it is false.

And then we are also looking at this string operations if you have a string expression, the first string colon the second string. So, let us look at how is this specified you can see here, expression, string colon, and then the regular expression. So, I am placing the string here in quotes, because if you do not do that, then the expression will take only Octavio as a first part of the string.

And then it is expecting some command after that like a colon, but then something else has come. So, it will have a problem. So, we have quoting the string within double quotes here, so that that part will not be having an error and then the regular expression is shown here. What is the regular expression? I am expecting a word at the beginning of the string to start error with a small letter or a capital letter and it should be sounding like either octave or Octavio.

So, both will be accepted. Now, this regular expression search with the `expr` is anchored so at least it will be searching only from the beginning of the string. So, you should that search for something which is in the part of the string but not but only at the beginning of the string.

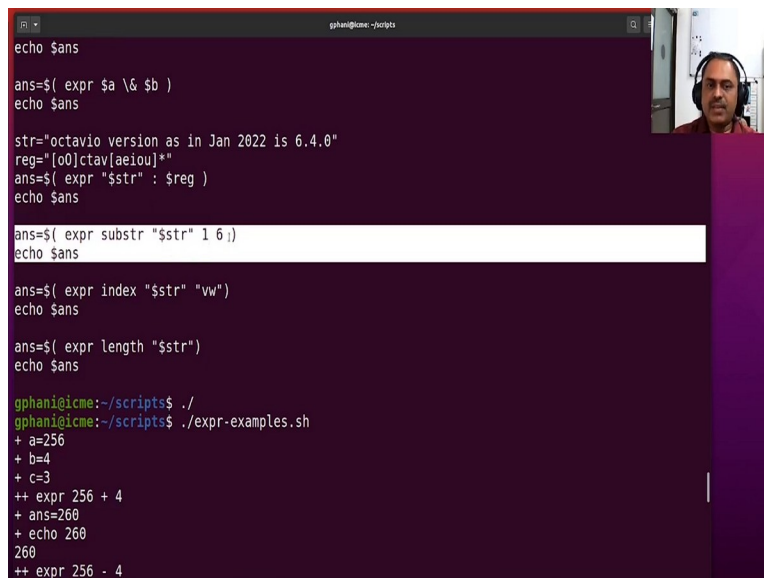
(Refer Slide Time: 18:44)



```
gphani@cme: ~/scripts
+ echo 1
1
++ expr 256 '|' 4
+ ans=256
+ echo 256
256
++ expr 256 '&' 4
+ ans=256
+ echo 256
256
+ str='octavio version as in Jan 2022 is 6.4.0'
+ reg='[o0]ctav[aeiou]*'
++ expr 'octavio version as in Jan 2022 is 6.4.0' : '[o0]ctav[aeiou]*'
+ ans=7
+ echo 7
7
++ expr substr 'octavio version as in Jan 2022 is 6.4.0' 1 6
+ ans=octavi
+ echo octavi
octavi
++ expr index 'octavio version as in Jan 2022 is 6.4.0' vw
+ ans=5
+ echo 5
5
++ expr length 'octavio version as in Jan 2022 is 6.4.0'
+ ans=39
+ echo 39
39
gphani@cme:~/scripts$
```

So, when we did that, you can see that when I am asking it to match you can see Octavio has been matched and you can see the 7 characters so the answer is 7. So, it is telling that up to the 7th character, there is a match available with the regular expression that is provided. A regular expression provided is within the limits of the BRE engine. So, no plus sign for example. And it is fitting what is there in the beginning of the string. So, the answer is 7.

(Refer Slide Time: 19:14)



```
gphani@cme: ~/scripts
echo $ans

ans=$( expr $a \& $b )
echo $ans

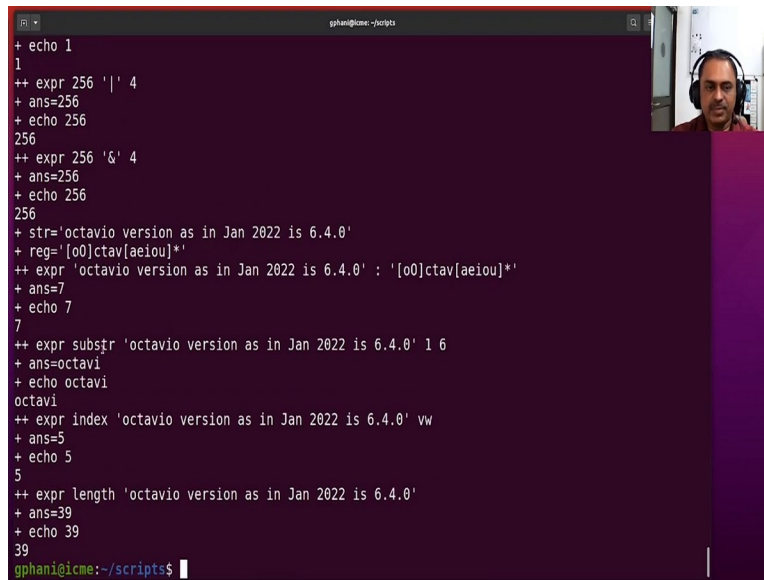
str="octavio version as in Jan 2022 is 6.4.0"
reg="[o0]ctav[aeiou]*"
ans=$( expr "$str" : $reg )
echo $ans

ans=$( expr substr "$str" 1 6 :)
echo $ans

ans=$( expr index "$str" "vw")
echo $ans

ans=$( expr length "$str")
echo $ans

gphani@cme:~/scripts$ ./
gphani@cme:~/scripts$ ./expr-examples.sh
+ a=256
+ b=4
+ c=3
++ expr 256 + 4
+ ans=260
+ echo 260
260
++ expr 256 - 4
```

A terminal window with a dark purple background. The window title is 'gphani@icme: ~/scripts'. It shows the execution of a shell script with various commands and their outputs. A small video inset in the top right corner shows a man with a beard and headphones. The script content is as follows:

```
+ echo 1
1
++ expr 256 '%' 4
+ ans=256
+ echo 256
256
++ expr 256 '&' 4
+ ans=256
+ echo 256
256
+ str='octavio version as in Jan 2022 is 6.4.0'
+ reg='[o0]ctav[aeiou]*'
++ expr 'octavio version as in Jan 2022 is 6.4.0' : '[o0]ctav[aeiou]*'
+ ans=7
+ echo 7
7
++ expr substr 'octavio version as in Jan 2022 is 6.4.0' 1 6
+ ans=octavi
+ echo octavi
octavi
++ expr index 'octavio version as in Jan 2022 is 6.4.0' vw
+ ans=5
+ echo 5
5
++ expr length 'octavio version as in Jan 2022 is 6.4.0'
+ ans=39
+ echo 39
39
gphani@icme:~/scripts$
```

Next we are asking what is the substring? So, here I am asking what is the substring of whatever is provided as a string and starting from the first position and for the 6 characters. So, that would be here, if you see here substring of whatever string is provided starting from the first character up to 6 characters, so that would be just octavi and that has been returned. So, you can chop off a part of the string and return it.

Similarly, you can also ask index that is any of the two characters that are provided? Where is the first occurrence of any of those characters? So, if you look at from the beginning, the very first occurrence is v, which is at the 5th position so the answer is 5. So, you can actually expand these two character to something else also. And then the length of the string also can be measured using expr length followed by the string and the answer is 39. So, what we will do is now we will go and edit this script.

(Refer Slide Time: 20:16)

```
gphani@icme: ~/scripts
echo $ans

ans=$( expr $a = $b )
echo $ans

ans=$( expr $a != $b )
echo $ans

ans=$( expr $a \\\ $b )
echo $ans

ans=$( expr $a \& $b )
echo $ans

str="octavio version as in Jan 2022 is 6.4.0"
reg="[o0]ctav[aeiou]*)"
ans=$( expr "$str" : $reg )
echo $ans

ans=$( expr substr "$str" 8 8 )
echo $ans

ans=$( expr index "$str" "Js")
echo $ans

ans=$( expr length "$str")
echo $ans

:
```

```
gphani@icme: ~/scripts
octavi
++ expr index 'octavio version as in Jan 2022 is 6.4.0' vw
+ ans=5
+ echo 5
5
++ expr length 'octavio version as in Jan 2022 is 6.4.0'
+ ans=39
+ echo 39
39
gphani@icme:~/scripts$ vi expr-examples.sh
gphani@icme:~/scripts$ ./expr-examples.sh
260
252
1024
64
1
1
0
0
1
0
1
256
256
7
version
12
39
gphani@icme:~/scripts$
```

```
gphani@cme: ~/scripts
+ echo 256
256
++ expr 256 '&' 4
+ ans=256
+ echo 256
256
+ str='octavio version as in Jan 2022 is 6.4.0'
+ reg='[o0]ctav[aeiou]*'
++ expr 'octavio version as in Jan 2022 is 6.4.0' : '[o0]ctav[aeiou]*'
+ ans=7
+ echo 7
7
++ expr substr 'octavio version as in Jan 2022 is 6.4.0' 1 6
+ ans=octavi
+ echo octavi
octavi
++ expr index 'octavio version as in Jan 2022 is 6.4.0' vw
+ ans=5
+ echo 5
5
++ expr length 'octavio version as in Jan 2022 is 6.4.0'
+ ans=39
+ echo 39
39
gphani@cme:~/scripts$ vi expr-examples.sh
gphani@cme:~/scripts$ ./expr-examples.sh
260
252
1024
```

```
gphani@cme: ~/scripts
260
252
1024
64
1
1
0
0
1
0
1
256
256
7
version
12
39
gphani@cme:~/scripts$ tail expr-examples.sh

ans=$( expr substr "$str" 8 8 )
echo $ans

ans=$( expr index "$str" "Js" )
echo $ans

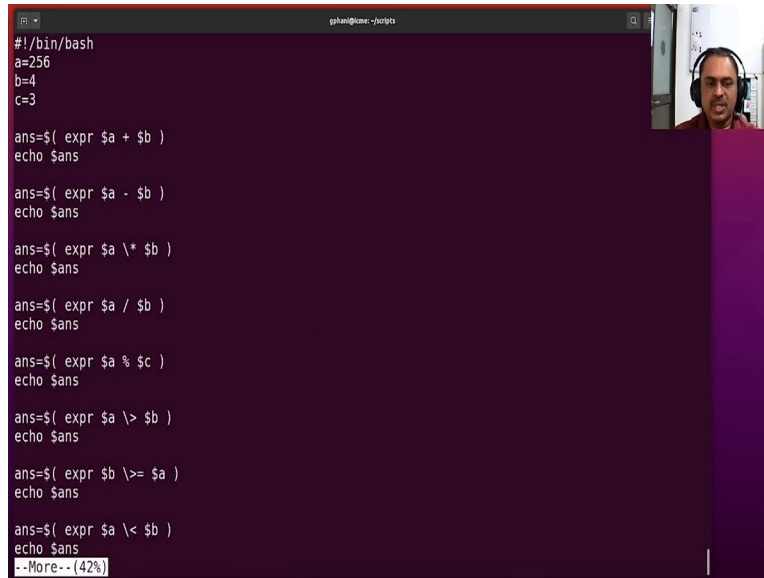
ans=$( expr length "$str" )
echo $ans
gphani@cme:~/scripts$
```

And we remove the debugging option, we do not need that. And here what we would do is look for the strings which are slightly different. So, I would put at is as j and then s j and s. And here, instead of starting with 1, I would start with 8, and to go up to 8 characters. So, let us go ahead and try this out. And you will see that the answer here is version. So, if you see here, when we start when we start at the eighth character, then version followed by a space is the answer and that has been returned here.

And then, look at the last command so Js, I am asking the position and if you look at the J position and s position, the very first occurrence is actually of s. And that would occur here, for example, this is 7 8 9 10 11 12. So, it is occurring at the twelfth position so, you can see that so,

the answer is 12. And then, the last thing is about the length of the string that is 39 same as what we have seen earlier.

(Refer Slide Time: 21:34)



```
#!/bin/bash
a=256
b=4
c=3

ans=$(( expr $a + $b ))
echo $ans

ans=$(( expr $a - $b ))
echo $ans

ans=$(( expr $a \* $b ))
echo $ans

ans=$(( expr $a / $b ))
echo $ans

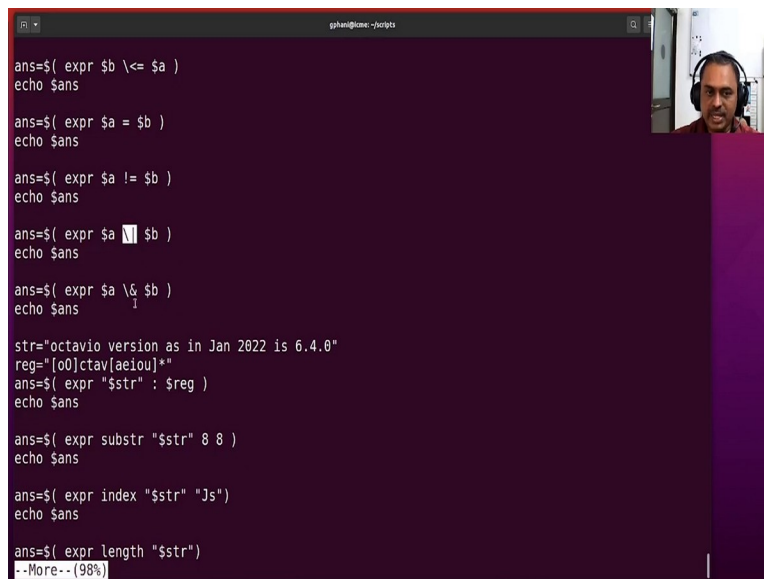
ans=$(( expr $a \% $c ))
echo $ans

ans=$(( expr $a \> $b ))
echo $ans

ans=$(( expr $b \>= $a ))
echo $ans

ans=$(( expr $a \< $b ))
echo $ans
--More-- (42%)
```

The terminal window displays a series of arithmetic and comparison operations using the `expr` command. It starts with variable assignments for `a`, `b`, and `c`, followed by calculations for addition, subtraction, multiplication, division, and modulus. It also includes greater-than, greater-than-or-equal-to, and less-than comparisons. The output shows the results of these operations, and the prompt indicates that the content is truncated with `--More-- (42%)`.



```
ans=$(( expr $b \<= $a ))
echo $ans

ans=$(( expr $a = $b ))
echo $ans

ans=$(( expr $a != $b ))
echo $ans

ans=$(( expr $a \& $b ))
echo $ans

ans=$(( expr $a \& $b ))
echo $ans

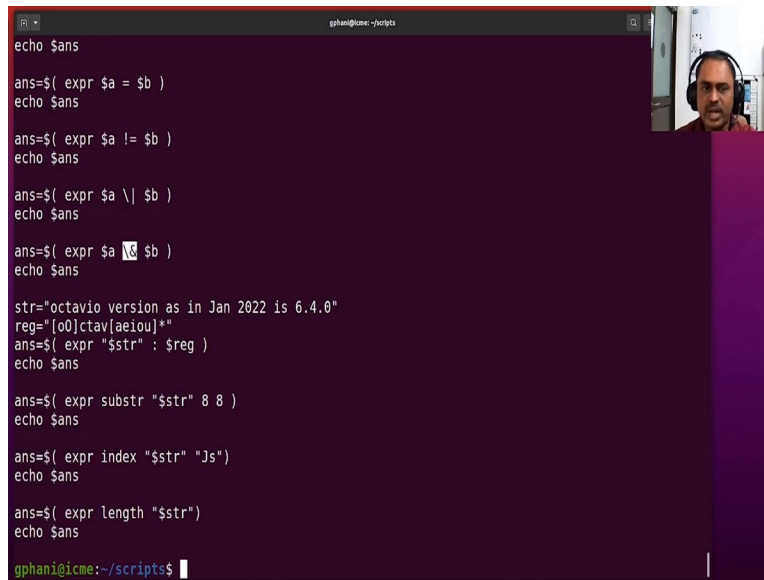
str="octavio version as in Jan 2022 is 6.4.0"
reg="[o0]ctav[aeiou]*"
ans=$(( expr "$str" : $reg ))
echo $ans

ans=$(( expr substr "$str" 8 8 ))
echo $ans

ans=$(( expr index "$str" "Js" ))
echo $ans

ans=$(( expr length "$str" ))
echo $ans
--More-- (98%)
```

The terminal window continues with string operations. It shows comparisons for less-than-or-equal-to, equality, and inequality. It also demonstrates bitwise AND operations. A string `str` is defined, and a regular expression `reg` is used to match a substring. The `expr` command is used to extract a substring and find the index of a character. Finally, the length of the string is calculated. The prompt indicates that the content is truncated with `--More-- (98%)`.



```
gphani@icme: ~/scripts
echo $ans

ans=$( expr $a = $b )
echo $ans

ans=$( expr $a != $b )
echo $ans

ans=$( expr $a \| $b )
echo $ans

ans=$( expr $a \> $b )
echo $ans

str="octavio version as in Jan 2022 is 6.4.0"
reg="[oO]ctav[aeiou]*"
ans=$( expr "$str" : $reg )
echo $ans

ans=$( expr substr "$str" 8 8 )
echo $ans

ans=$( expr index "$str" "Js")
echo $ans

ans=$( expr length "$str")
echo $ans

gphani@icme:~/scripts$
```

So, just let this script once more. So, you can see that we are able to check the various operators. So, some operators are slightly especially given so that we escaped the meaning. So, you also escaped the meaning of the greater than symbol, as well as the less than symbol because they have special meaning shell. So we have put those backslashes there. And the pipe also has a special meaning ampersand also has special meaning so we are also escaping them.

Other than that, by and large, the syntax of `expr` is exactly how I have shown in the slides. So, you can see that fairly rich set of operators are available using `expr` command. And also numerics are possible with 4 different syntaxes. So, you can go ahead and try that out as part of your shell scripts.

(Refer Slide Time: 22:21)

here doc feature



```
a=2.5
b=3.2
c=4
d=$(bc -l << EOF
scale = 5
($a+$b)^$c
EOF
)
echo $d
```

1055.6001

Marker designation need not be EOF

```
a=2.5
b=3.2
c=4
d=$(bc -l <<- ABC
    scale = 5
    ($a+$b)^$c
    ABC
)
echo $d
```

A hyphen tells bash to
ignore leading tabs

Now there is a interesting feature called heredoc feature. What it means is that when you have the requirement to give a long string, with special characters inside, or maybe line breaks, et cetera, then you do not have to struggle by introducing the backslash n, or quoting and unquoting or escaping some special characters et cetera. So, you can just blindly type out whatever you want to assign as a part of the long string. And have a marker so that the input is taken up to the marker.

So, on the left hand side, you have seen that the marker is EOF, which is a default, if you do not say anything, that means that EOF is what it is expecting up to the line in which the first character is E. And then, the word is EOF the input is read, and that input is then fed at the location that you are supposed to be giving. So, bc minus l followed by whatever is the text is supposed to be inputted to the bc that is actually given here.

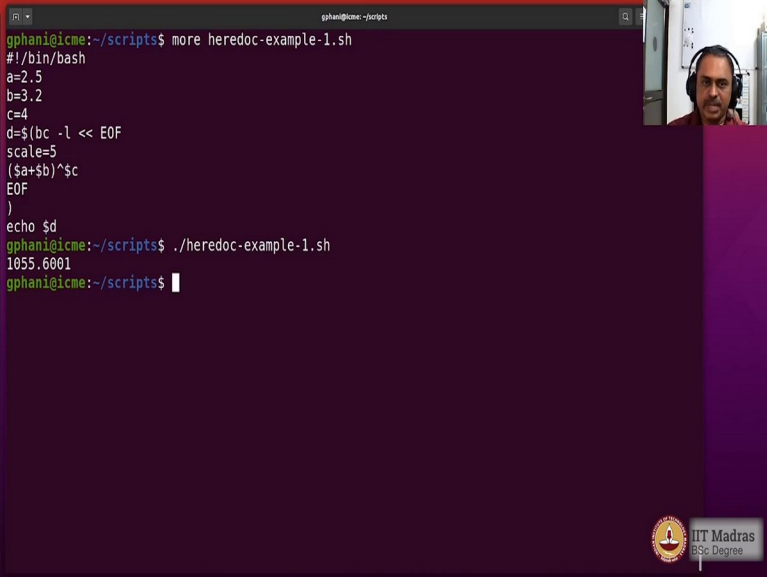
Scale is corresponding to the precision of calculation. And the operation that is done by bc is actually take the sum of the first two numbers and raise it to the third number. So, you can see this upload floating point operation. And we are able to do that within the shell script and getting a floating point answer by using this here doc feature so that our code is readable. Now, sometimes you may want to make the code readable, slightly better by using indentation.

So, if you want, you can actually have a tab character to indent the script. But you need to actually have a hyphen after that to less than symbols for the marker so that the tab characters are

ignored. So, this is important because there are some commands this in this case it is bc, which does not bother about the tab characters, but there could be some commands where the initial tab could be a problem.

So, if you want that to be ignored, then it is important for you to mention that there is a minus sign there so that these are ignored. And then the marker actually can be anything else. So, you can actually write it as EOF or anything else. And you can indent the marker also because you have actually given here minus sign. So, both these scripts will do the same feature and we can just try that out right away.


(Refer Slide Time: 24:35)

A terminal window with a dark purple background. The prompt is 'gphanigme:~/scripts'. The user enters 'more heredoc-example-1.sh'. The script content is displayed: '#!/bin/bash', 'a=2.5', 'b=3.2', 'c=4', 'd=\$(bc -l << EOF', 'scale=5', '(\$a+\$b)^\$c', 'EOF', ')', 'echo \$d'. The user then enters './heredoc-example-1.sh'. The output '1055.6001' is shown. The prompt returns to 'gphanigme:~/scripts'.

```
gphanigme:~/scripts$ more heredoc-example-1.sh
#!/bin/bash
a=2.5
b=3.2
c=4
d=$(bc -l << EOF
scale=5
($a+$b)^$c
EOF
)
echo $d
gphanigme:~/scripts$ ./heredoc-example-1.sh
1055.6001
gphanigme:~/scripts$
```

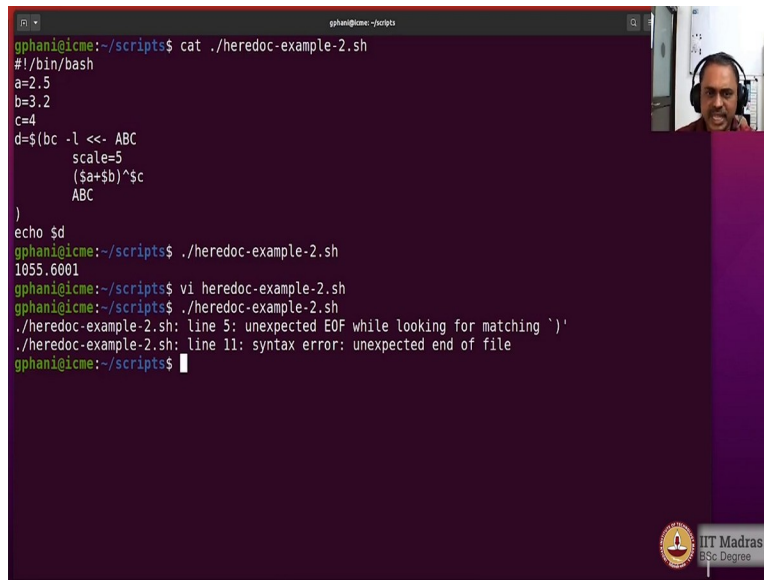
So, here is the example first example of here doc, where we are having the input given to the dc minus l going up to here and that is actually fed to the bc and look at the output of that and you see that the answer has come so the calculation has been performed within the back script. Now let us look at the second example also.

(Refer Slide Time: 24:59)




```
gphani@icme: ~/scripts
gphani@icme:~/scripts$ cat ./heredoc-example-2.sh
#!/bin/bash
a=2.5
b=3.2
c=4
d=$(bc -l <<- ABC
    scale=5
    ($a+$b)^$c
    ABC
)
echo $d
gphani@icme:~/scripts$ ./heredoc-example-2.sh
1055.6001
gphani@icme:~/scripts$
```

[illegible]

A terminal window titled 'gphanigme: ~/scripts' displays the following commands and output:

```
gphanigme:~/scripts$ cat ./heredoc-example-2.sh
#!/bin/bash
a=2.5
b=3.2
c=4
d=$(bc -l <<- ABC
    scale=5
    ($a+$b)^$c
    ABC
)
echo $d
gphanigme:~/scripts$ ./heredoc-example-2.sh
1055.6001
gphanigme:~/scripts$ vi heredoc-example-2.sh
gphanigme:~/scripts$ ./heredoc-example-2.sh
./heredoc-example-2.sh: line 5: unexpected EOF while looking for matching `)'
./heredoc-example-2.sh: line 11: syntax error: unexpected end of file
gphanigme:~/scripts$
```



So, here we have used the here doc with a different marker. So the marker is not EOF but it is abc, which is a load. And we have also used a minus symbol there a hyphen to indicate that we are actually indenting the text. So, that it is not expecting to see abc at the beginning of the line, but with the same indentation and this is going to be also accepted. So, let us execute that. And you see that the same answer as what we have seen.

Now, let us go ahead and edit this. So, that the error will be shown. So, I forget to include the minus option there minus the hyphen symbol after the double less than and you see that there is no syntax highlighting that is happening for abc already a hint that it is not likely to work. Let us go ahead and try that out. And you see that there is a syntax error because it was expecting to see the marker, and it was not finding that.

(Refer Slide Time: 26:03)

[illegible]

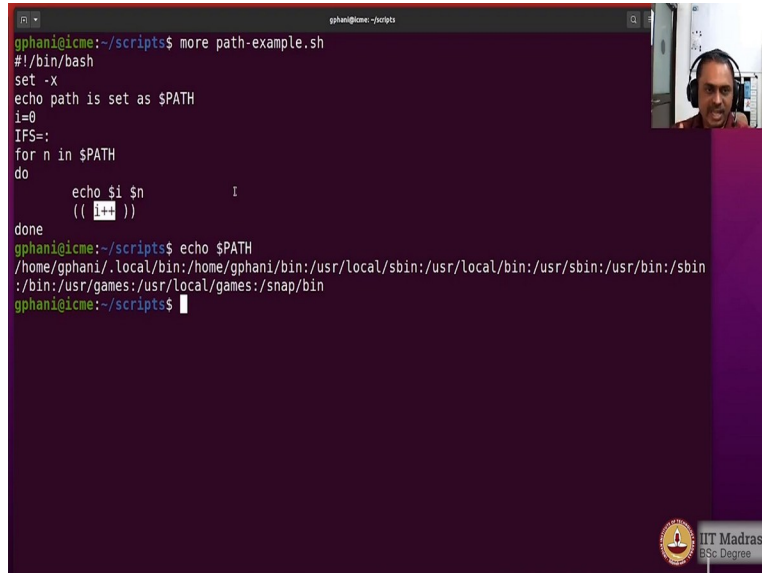
And when we put that minus sign, then syntax highlighting also shows that everything is fine. Now the marker also we could change, so I will change it to say BLAH. And go on to execute that. And you see that that parts are quite fine. So, you can see that the marker is very useful in keeping the code quite neat and also indented. So, that you can actually give a multi line input to any command and using the command substitution we are getting the output of the bc command and then writing it to b variable and then printing that variable out.

So, you can perform floating point operations within the shell. Now, we have done some arithmetic and we have also seen how this can be done. We have also seen how you can do some

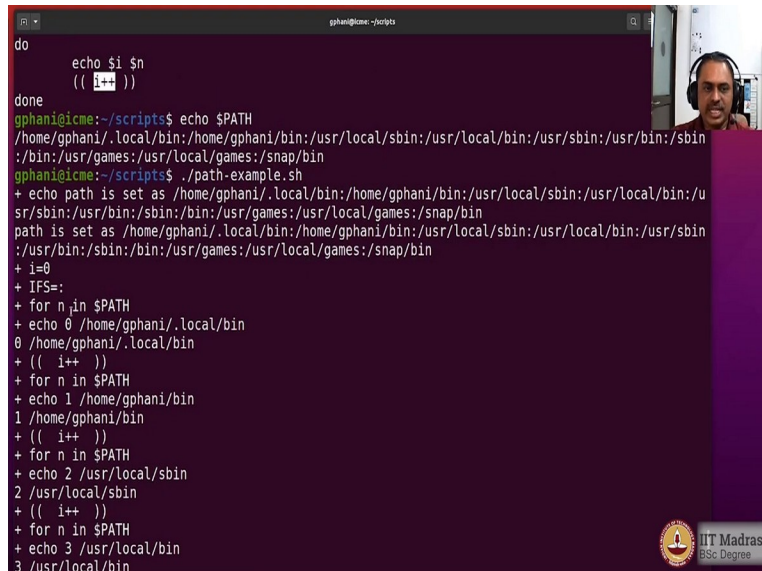
special operations like increment. So, let us go ahead and see whether we can explore the so called path variable, because it is a very important variable. Because as I mentioned, we should keep our shell scripts in some specific directory.

So, if it is available in the path is good, you can use the commands without the path prepending this script name.

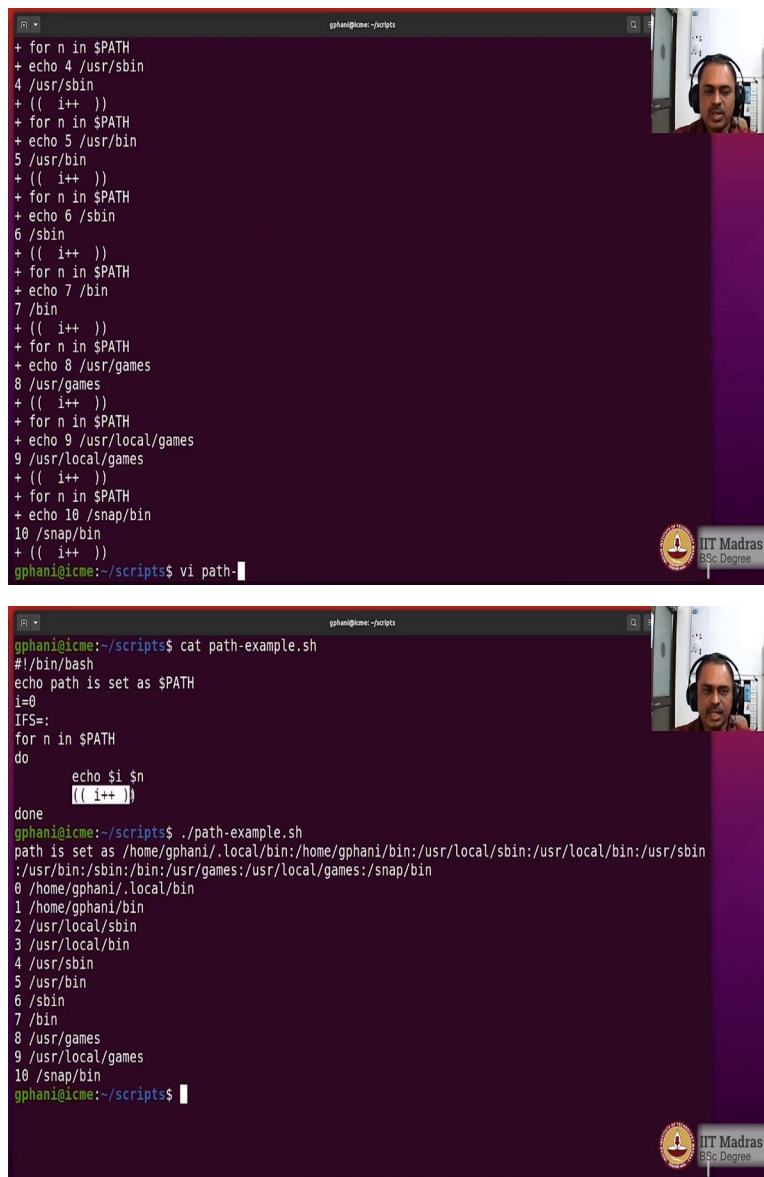
(Refer Slide Time: 27:20)



```
gphani@icme:~/scripts$ more path-example.sh
#!/bin/bash
set -x
echo path is set as $PATH
i=0
IFS=:
for n in $PATH
do
    echo $i $n          I
    (( i++ ))
done
gphani@icme:~/scripts$ echo $PATH
/home/gphani/.local/bin:/home/gphani/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin
gphani@icme:~/scripts$
```



```
do
    echo $i $n
    (( i++ ))
done
gphani@icme:~/scripts$ echo $PATH
/home/gphani/.local/bin:/home/gphani/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/snap/bin
gphani@icme:~/scripts$ ./path-example.sh
+ echo path is set as /home/gphani/.local/bin:/home/gphani/bin:/usr/local/sbin:/usr/local/bin:/u
sr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
path is set as /home/gphani/.local/bin:/home/gphani/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin
:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
+ i=0
+ IFS=:
+ for n in $PATH
+ echo 0 /home/gphani/.local/bin
0 /home/gphani/.local/bin
+ (( i++ ))
+ for n in $PATH
+ echo 1 /home/gphani/bin
1 /home/gphani/bin
+ (( i++ ))
+ for n in $PATH
+ echo 2 /usr/local/sbin
2 /usr/local/sbin
+ (( i++ ))
+ for n in $PATH
+ echo 3 /usr/local/bin
3 /usr/local/bin
```



The image contains two terminal window screenshots. The top screenshot shows a script being edited in a vi editor. The script iterates through the directories in the \$PATH variable and prints each one. The bottom screenshot shows the script being executed, which prints the contents of \$PATH as a list of directories separated by newlines.

```
gphani@cme: ~/scripts
+ for n in $PATH
+ echo 4 /usr/sbin
4 /usr/sbin
+ (( i++ ))
+ for n in $PATH
+ echo 5 /usr/bin
5 /usr/bin
+ (( i++ ))
+ for n in $PATH
+ echo 6 /sbin
6 /sbin
+ (( i++ ))
+ for n in $PATH
+ echo 7 /bin
7 /bin
+ (( i++ ))
+ for n in $PATH
+ echo 8 /usr/games
8 /usr/games
+ (( i++ ))
+ for n in $PATH
+ echo 9 /usr/local/games
9 /usr/local/games
+ (( i++ ))
+ for n in $PATH
+ echo 10 /snap/bin
10 /snap/bin
+ (( i++ ))
gphani@cme:~/scripts$ vi path
```



```
gphani@cme:~/scripts$ cat path-example.sh
#!/bin/bash
echo path is set as $PATH
i=0
IFS=:
for n in $PATH
do
    echo $i $n
    (( i++ ))
done
gphani@cme:~/scripts$ ./path-example.sh
path is set as /home/gphani/.local/bin:/home/gphani/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
0 /home/gphani/.local/bin
1 /home/gphani/bin
2 /usr/local/sbin
3 /usr/local/bin
4 /usr/sbin
5 /usr/bin
6 /sbin
7 /bin
8 /usr/games
9 /usr/local/games
10 /snap/bin
gphani@cme:~/scripts$
```

So, let us try that out. So, the PATH example is here. So what are we doing, we are actually setting the minus x option to see what is happening. And what we are doing here is IFS is equal to colon. So, we are actually cell setting the internal field separator value as colon. So, the PATH variable if you see, the directories are all listed one after other by separating with a colon symbol.

So, I am saying that IFS is colon for n in PATH, so if is colon, which means that all these directory names are individual fields. So, then it is like an array and therefore, for n in the list of directories as an array separated by a colon what we are doing is we are actually printing a

number integer followed by the directory. And then we are actually incrementing that number by 1 each time we are printing.

So, from our knowledge of arithmetic operations, we know that `i` value is just going up 1 every time the loop is running, and then the editor name is being printed. So, we are actually able to see what is the sequence by which the `PATH` will be searched directly after directory. So, let us just check how does this work and you will see that the `minus x` option is actually telling you whatever it is doing and the very first is 0.

The very first directory is this one, the second one is this one, the third one is this one and so on. Now, I will go and disable that option and show you in a neat fashion how this option how the output looks like. So, here you can see that the script without the debugging option shows you that we are starting at 0 and some 0 onwards what is the sequence of directories this will be searched in the `PATH` array.

So, we have achieved this by using the arithmetic operation to print the number in the front and splitting the directories is done by `IFS` is equal to colon and then we are using the for loop as if the dollar `PATH` is basically an array almost. So, if you have `IFS` is equal to colon it is as good as an array. And then we are going and looking at those segments. So, you can do that with any a line that contains a specific field separator and then you can treat each of those segments as fields and inspect them and do some operation.