

System Commands
Professor. Gandham Phanikumar
Metallurgical and Materials Engineering
Indian Institute of Technology, Madras
Bash scripts - Part 2C

(Refer Slide Time: 00:14)

eval



```
eval my-arg
```

- Execute argument as a shell command
- Combines arguments into a single string
- Returns control to the shell with exit status




There is a feature called eval, which would evaluate the string that was given as if it is a command, and it will be passed on to the shell to execute. One warning, never evaluate a user supplied string on any shell. Because, such features if they are available in any script will make the system vulnerable to attacks because the user can supply any arbitrary command and if it gets executed and sometimes dangerously by a super user permission, then anything can happen.

So, it is a very good security practice to not evaluate any user supplied string as it is. With that warning, you can go ahead and see whether you can construct commands and then evaluate them on the shell. So, sometimes, there are some commands which you would like to construct using various string operations, et cetera.

And finally, you have got a set of strings, then all the strings can be treated as a single string and then getting executed on the shell using the eval command. And once it is over, the control will be given back to the shell, which has launched the eval command.

(Refer Slide Time: 01:26)

```
gphani@icme:~/scripts$ cat eval-example.sh
#!/bin/bash
cmd="date"
fmt="%d-%B-%Y"
eval $cmd $fmt
gphani@icme:~/scripts$ ./eval-example.sh
25-January-2022
gphani@icme:~/scripts$ vi eval-example.sh
gphani@icme:~/scripts$ cat eval-example.sh
#!/bin/bash
cmd="date"
fmt="%d-%B-%Y"
echo $$
eval $cmd $fmt; echo $$
gphani@icme:~/scripts$ echo $$
2360
gphani@icme:~/scripts$ ./eval-example.sh
12809
25-January-2022
12809
gphani@icme:~/scripts$
```




```
gphani@icme:~/scripts$ cat function-example.sh
#!/bin/bash
usage()
{
    echo usage $1 str1 str2
}

swap()
{
    echo $2 $1
}

if [ $# -lt 2 ]
then
    usage $0
    exit 1
fi

swap $1 $2
gphani@icme:~/scripts$ ./function-example.sh
usage ./function-example.sh str1 str2
gphani@icme:~/scripts$ ./function-example.sh arg1
usage ./function-example.sh str1 str2
gphani@icme:~/scripts$ ./function-example.sh arg1 arg2
arg2 arg1
gphani@icme:~/scripts$ cp function-example.sh mylib.sh
gphani@icme:~/scripts$ vi mylib.sh
```



And so which means that this was run in the same shell as the eval example dot sh script, sometimes, it is useful to have functions written in your script, so that certain specific actions are taken care by respective functions. And you can reuse those functions by keeping them in a particular script file and then sourcing it in every script that you would write so that standard functions are available in multiple scripts.

And this can be things like writing an error message or logging information to a log book, providing help documentation about the script et cetera. So, here is an example of a script which has some functions. So, what we have is two functions one function usage, the function usage, what it does is it will take the argument that was passed to it and tell that the script has to be used in a particular format, that is the name by which it has been launched.


Here, you can see that I am calling it with the dollar 0, which means the what is the name by which the script has been called. And then it should be called with the 2 strings one after the other. So, I am telling a help information on the screen, how the script has to be launched. Now, there is another function which this swapping.

So, it is printing the two variables that are passed in a reverse order. Now you can see the dollar 1 and dollar 2 do not correspond to the arguments to the shell. But arguments to the call of the swap function. The same thing is applicable here. Dollar 1 is the argument passed to the usage call.

So, here, if you see, in case the script has been called with less than two arguments, then I am printing the usage document and I am exiting, and in case it has been called with two arguments or more than I am swapping the first and second arguments. So, you can see here the name of the script has been passed to usage and there is a first argument being sent to usage and therefore, dollar 1 corresponds to the dollar 0, which is the name of the script that has been used for launching.


So, let us try this out. So, I try without any arguments. So, you can see that the usage has been displayed and you can see that the dollar 1 here is dollar 0, which is the name of the script and it is quite nice because I do not have to hard code that inside. And str1, str2 is what is here saying that you have to use this particular script with two string has arguments.

So, here is an example where some of the actions are packaged away in functions and these can also be kept in a separate file to source. So, let us see that also now. So, what I would do is that function example, I would say, mylib dot sh and I go to mylib dot sh and I will have these functions that are kept in the library.



```
gphani@icme: ~/scripts
gphani@icme:~/scripts$ cat mylib.sh
usage()
{
    echo usage $1 str1 str2
}

swap()
{
    echo $2 $1
}
gphani@icme:~/scripts$ vi function-example.sh
```



```
ghani@icmr:~/scripts
./bin/bash
usage()
{
    echo usage $1 str1 str2
}

swap()
{
    echo $2 $1
}

if [ $# -lt 2 ]
then
    usage $0
    exit 1
fi

swap $1 $2

~
~
~
~
~
~
~
~
~
~

"function-example.sh" 18L, 128C
```


So, that they are not seen in the main script file to keep your code elegant and also to perhaps not touch the functions that are well written. And therefore, they do not have to be disturbed when you are editing.

(Refer Slide Time: 07:08)

getopts

```
while getopts "ab:c:" options;
do
    case "${options}" in
        b)
            barg=${OPTARG}
            echo accepted: -b $barg
            ;;
        c)
            carg=${OPTARG}
            echo accepted: -c $carg
            ;;
        a)
            echo accepted: -a
            ;;
        *)
            echo Usage: -a -b barg -c carg
            ;;
    esac
done
```

This script can be invoked with only three options: **a**, **b**, **c**. The options **b** and **c** will take arguments.

Now as you have seen that, we can pass arguments to shell scripts. And it is a good idea to have some of the arguments designed in the way that Linux commands are also working. So, you can pass on some options, and then have a way by which those options are processed in an automatic fashion. Now, getopts is a feature available within the bash so that this processing of options passed to the script is done in a very elegant manner.

So, for example, in this script, you can call the script using the options a b or c that is minus a minus b and minus c. And if you are giving options like minus b or minus c, then they should have a value after that, so minus b, then b argument minus c, then c argument. However, minus a will not require any argument.

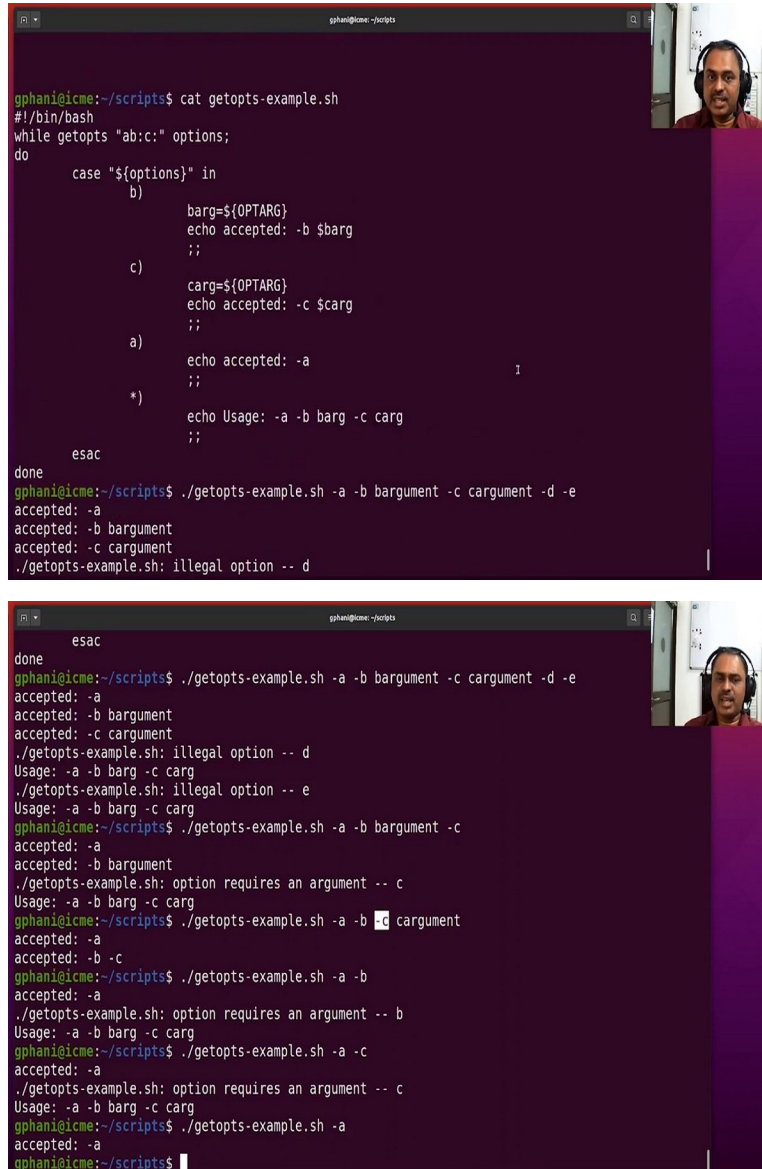
So, the colon following the letter shows that that particular option will have a value that is being expected. So, some kind of a argument is being expected for those options. This is processed very elegantly here, saying that with a case loop. So, if the options are such that the b is option, then the argument that is supplied with the b is then put in a variable called barg.

And it is the c option, then the argument that is supplied is put in the carg. And if it is a, then you do not have to do any of those. And by default, you can tell the usage pattern of that particular

script and then close the case loop and close the do loop. So, this is a very neat way by which the options are being processed and you do not have to now worry about which position the a b c options have come because the case loop will take care of those kinds of actions.

So, let us see this usage in a script so that you can try this out for any professional written script that you would like to do shortly.

(Refer Slide Time: 09:01)



```
gphani@cme:~/scripts$ cat getopt-example.sh
#!/bin/bash
while getopt "ab:c:" options;
do
    case "${options}" in
        b)
            barg=${OPTARG}
            echo accepted: -b $barg
            ;;
        c)
            carg=${OPTARG}
            echo accepted: -c $carg
            ;;
        a)
            echo accepted: -a
            ;;
        *)
            echo Usage: -a -b barg -c carg
            ;;
    esac
done
gphani@cme:~/scripts$ ./getopts-example.sh -a -b bargument -c cargument -d -e
accepted: -a
accepted: -b bargument
accepted: -c cargument
./getopts-example.sh: illegal option -- d

gphani@cme:~/scripts$ ./getopts-example.sh -a -b bargument -c cargument -d -e
accepted: -a
accepted: -b bargument
accepted: -c cargument
./getopts-example.sh: illegal option -- d
Usage: -a -b barg -c carg
./getopts-example.sh: illegal option -- e
Usage: -a -b barg -c carg
gphani@cme:~/scripts$ ./getopts-example.sh -a -b bargument -c
accepted: -a
accepted: -b bargument
./getopts-example.sh: option requires an argument -- c
Usage: -a -b barg -c carg
gphani@cme:~/scripts$ ./getopts-example.sh -a -b -c cargument
accepted: -a
accepted: -b -c
gphani@cme:~/scripts$ ./getopts-example.sh -a -b
accepted: -a
./getopts-example.sh: option requires an argument -- b
Usage: -a -b barg -c carg
gphani@cme:~/scripts$ ./getopts-example.sh -a -c
accepted: -a
./getopts-example.sh: option requires an argument -- c
Usage: -a -b barg -c carg
gphani@cme:~/scripts$ ./getopts-example.sh -a
accepted: -a
gphani@cme:~/scripts$
```

So, here is a very simple script that we have just not seen. So, we are going to launch by giving the options. So, let us give the options as minus a minus b, then b argument minus c, c argument

and let us say minus d minus e. I am giving some extra options. Now, what will happen is that the case loop has worked and it is accepting minus a minus b argument minus c argument.

But it is also telling something more which is very useful that is it says that you have given an illegal option because the options that are accepted are a b c as given in the getopts. So, any other option is then illegal. So, it is also helping you to reduce the amount of coding because you do not now need to search for any other pattern and then throw some error. So, these errors are thrown by getopts which is very elegant.

Now let us skip those and let us say the c option is given without any argument. So, you see that again there is a requirement of an argument for the c option and it is being alerted. So, which means that again here the colon that is after the c is doing the entire magic of searching for an argument after the c option and if it is not there, then it is throwing an error.

Similarly, I can say c argument, but I skipped the b argument and you see what happens, what happens is that minus c is used as if it is argument for minus b after that the rest of it is ignored. And therefore, there are two options that are active minus a minus b. So, minus c is used as if it is an argument for b. Now, I would also be able to try that out for an error by skipping it.

So, you can see that there is nothing after b and therefore, what is expected as argument for the b option is missing and therefore, there is an error. I could now also see that for c, so the same error will be shown. Now, if I do not give any of those, there is no problem because there are only options, and therefore, this error, so you can see that the option processing and expecting an argument.

And throwing errors, if they are not available as per requirement is all done quite elegantly by using the getopts. By going through the getopts documentation. And implementing that in your scripts, you can make your script quite professional almost like any other Linux command.

(Refer Slide Time: 11:32)

select loop



```
echo select a middle one
select i in {1..10}
do
    case $i in
        1 | 2 | 3)
            echo you picked a small one;;
        8 | 9 | 10)
            echo you picked a big one;;
        4 | 5 | 6 | 7)
            echo you picked the right one
            break;;
    esac
done
echo selection completed with $i
```

Text menu !

Now one last thing that we would like to look at in the bash features is to provide a menu. So, let us say we have a set of things that user has to pick up. And a menu is to be shown. And the user has to pick one of those that are shown in the middle. So, this entire idea of showing a set of options as a menu, and then checking for the user input, seeing if the user input is one of the menu items that are listed. If it is not, then throwing an error.

And if it is matching, then assigning that particular value of the chosen menu to a variable, and then moving on so the interaction can be captured by using a select loop. Here is an example of a select loop. And let us go and explore this by running it. And this is a beautiful way by which a text menu can be provided within the terminal enrollment.

This is quite elegant, because the construction of select loop is quite simple. Like any other case loop, however, there is a additional action that is being performed, namely, whatever you choose is being matched with what is in the menu. And then unless it matches, the loop is continuing. So, that the user input is sought for only one of the listed menu items and not anything else. So, let us check that out how it works.

(Refer Slide Time: 12:54)

```
gphani@cme:~/scripts$ ./select-example.sh
select a middle one
1) 1
2) 2
3) 3
4) 4
5) 5
6) 6
7) 7
8) 8
9) 9
10) 10
#? 1
you picked a small one
#? 9
you picked a big one
#? 5
you picked the right one
selection completed with 5
gphani@cme:~/scripts$
```

```
gphani@cme:~/scripts$ ./select-example.sh
select a middle one
1) 1
2) 2
3) 3
4) 4
5) 5
6) 6
7) 7
8) 8
9) 9
10) 10
#? ksdkfjsdf
#? khkhsd
#? l;lksdf
#? 090923r
#? 1
you picked a small one
#? 2
you picked a small one
#? 3
you picked a small one
#? 4
you picked the right one
selection completed with 4
gphani@cme:~/scripts$
```

```
gphani@cme: ~/scripts
6) 6
7) 7
8) 8
9) 9
10) 10
#? ks)kdfjsdf
#? khkhsd
#? l;lkdsf
#? 090923r
#? 1
you picked a small one
#? 2
you picked a small one
#? 3
you picked a small one
#? 4
you picked the right one
selection completed with 4
gphani@cme:~/scripts$ cat select-example.sh
#!/bin/bash
echo select a middle one
select i in {1..10}
do
    case $i in
        1 | 2 | 3)
            echo you picked a small one
            ;;
        8 | 9 | 10)
            echo you picked a big one
            ;;
    esac
done
```

```
gphani@cme: ~/scripts
#? 1
you picked a small one
#? 2
you picked a small one
#? 3
you picked a small one
#? 4
you picked the right one
selection completed with 4
gphani@cme:~/scripts$ cat select-example.sh
#!/bin/bash
echo select a middle one
select i in {1..10}
do
    case $i in
        1 | 2 | 3)
            echo you picked a small one
            ;;
        8 | 9 | 10)
            echo you picked a big one
            ;;
        4 | 5 | 6 | 7)
            echo you picked the right one
            break
            ;;
    esac
done
echo selection completed with $i
gphani@cme:~/scripts$
```

So, the select example that we have chosen is same as what is in the slides. And we will run it now. And you see that I have shown 10 numbers, and I am asking a number that is falling in the middle of the range. So, if I choose a 1, it says that I have picked a small one small number, if I choose 9, it says I have picked up a big number.

If I choose 5, then it says that I have picked the right one and then the variable that is running with the select loop is given a value 5 and then we are printing it out. So, you can see the script here. Select completed with dollar i. So, i is a variable for the select loop. And i has been assigned a value 5 as you can see here. The final selection that is accepted is used to assign the

value of the variable which is used for the selected loop. And then we move on to the remaining part of the code.

Now let us try the same thing with options that not listed. So, what happens when I give options, like for example, I just give some random string. So, you see that it does not leave me, I cannot leave. And I have to give options that are only part of the menu. And I can give that. And you see that it allows to finish only when I come out of the loop as per the script.

And what is that you can see here, I am able to come out of the loop using break only when I choose either 4 or 5 or 6 or 7. Otherwise, I am not coming out of the loop. And that is what happened here. I picked 1 and 2 and 3. The loop is still continuing the moment I picked 4 I am able to come out of the loop. So, that is what has happened here.

So, you can use the break command to indicate what is the value or the pattern that you are looking for within the menu. Otherwise, you can ask the user to keep iterating and you can also be sure that if the user gives any input other than what is listed in this menu items, then it will be ignored. So, that way it is very elegant.

(Refer Slide Time: 15:07)



*You have seen most of the features needed to
create a professional bash script*

Explore by trying out !



So, you have now seen most of the features that are required to create a professional bash script. Please make sure that you follow some security features do not give set UID permission to the scripts unless what you are doing. And do not process user inputs to run commands because they

could contain some malicious code. Check for validity of the user input before you use in your scripts so that the script would run without any error.

And soon you can see that by packaging some of these actions as the library calls that are kept in a separate file. You can make a very professional looking bash script all by yourself and create new commands for your own processing of textual information on the Linux operating system. So, enjoy exploring by actually trying them out.