

System Commands
Professor. Gandham Phanikumar
Department of Metallurgical and Materials Engineering
Indian Institute of Technology, Madras
Overview of shell scripts

(Refer Slide Time: 0:14)

scripts



creating your own commands



Welcome to the session on scripts. In this session we will learn how to create our own commands. The experience you would have had by now using the text editors within the terminal would be of great help because you can open a terminal, write your script, edit it, save it, come out and then test it out right there within the same terminal.

(Refer Slide Time: 0:40)

Software Tools Principles



- Do one thing well
- Process lines of text, not binary
- Use regular expressions
- Default to standard I/O
- Don't be chatty
- Generate same output format accepted as input
- Let someone else do the hard part
- Detour to build specialized tools

Ref: Classic Shell Scripting – Arnold Robbins & Nelson H.F. Beebe



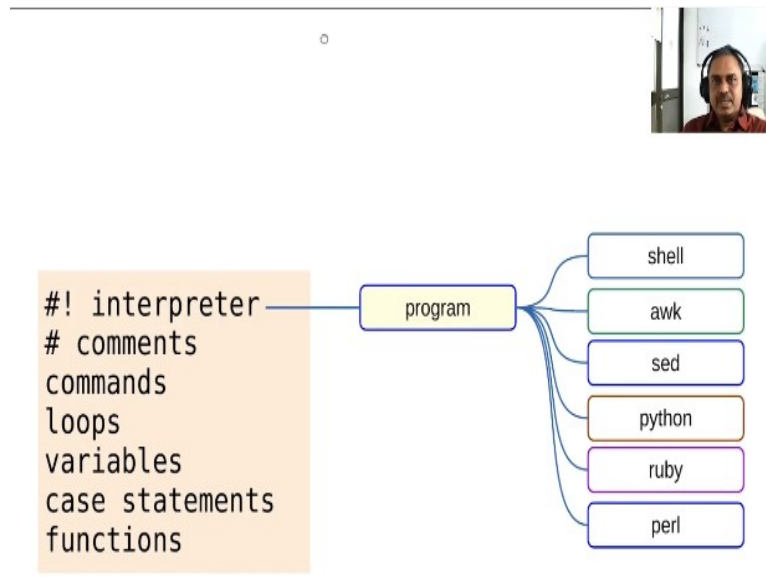
So, there are certain basic principles about software development that I borrowed from the book that is referred at the bottom and these principles guide the way Linux operating system has been developed over the years. So, each command is designed to do one thing very well and the commands usually process lines of text and not very often binary format.

And usually regular expressions are encouraged to be used, so that the scripts that we write are applicable for a wide variety of input text. And any output that has to be given should be given to the standard output and any input that has to be read can be read from the standard input so that these commands can be used along with pipe to combine with other functions.

Do not be chatty is a principle about the way most of the Linux commands work, that is if there is a command that works fine, then there is nothing to display on the screen and the output should be of the same format as what is accepted as the input. And if there are specific tasks that are required to combine various things, then leave it to the user so that the application that we are developing is to do just one thing very well and in the most generic fashion as possible.

And if you have a requirement to make a specialized tool, then use a detour so that you could make it for the specific need rather than working hard to make a specialized tool that would also be very generic because these are very different kind of viewpoints.

(Refer Slide Time: 2:36)



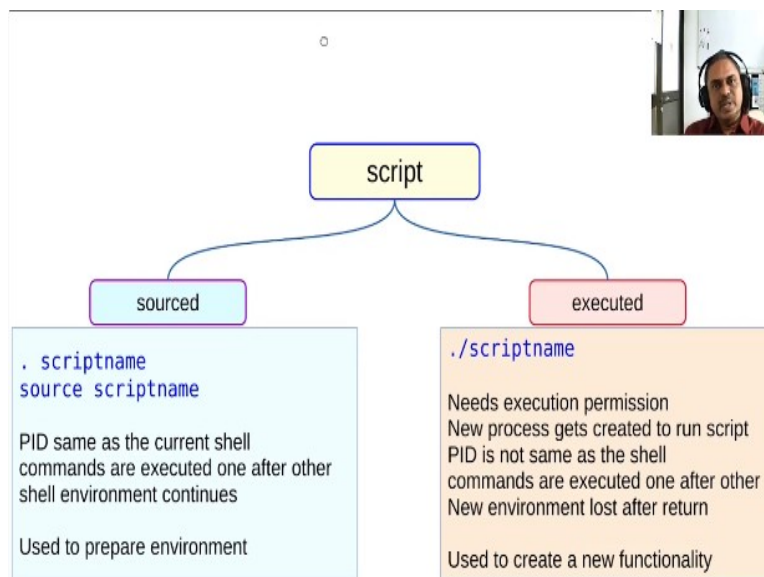
When you say script it need not be only a shell script okay, it can be a script that would have a series of commands that could be interpreted by any program which works as an interpreter and as we have known by now particularly with respect to the python which is an interpreted language, there are many languages that would act as an interpreter and therefore, scripts can be shell scripts or awk scripts or sed scripts or python scripts or ruby scripts or perl scripts and so on.

In our course we will be dealing with awk and sed also in the upcoming sessions and the remaining languages would be dealt in other courses where those languages will be put to greater use in application development. Awk and sed are included in all flavors of Linux and Unix and are integral parts of the operating system and therefore combining them with bash scripts is particularly an attractive proposition to have a solution that would work entirely natively within Linux environment.

A typical text that forms the script is shown on the left where the very first line is also actually a comment because all lines that start with the hash are comments but the very first line is special so there is an exclamation mark after the hash which indicates that following that exclamation mark is the path of the executable which would work as an interpreter for the rest of the text that is in the script file.

And therefore, if you have had in the first line hash bang slash bin slash bash it means that it is a bash script and similarly for all other scripts. And the shell scripts would contain commands which you have learnt already by now several commands and they would also contain typical language construction such as loops, variables, case statements and functions and so on.

(Refer Slide Time: 4:42)



Shell scripts are broadly categorized as two types, the ones which are supposed to be sourced and the ones which are supposed to be executed. Needless to say every script that is supposed to be sourced can also be executed but the difference lies in the way we invoke and the effect that it would have.

A sourced script is executed using either a dot or the keyword source followed by the name of the script and lines of commands will be read from this file and they would be executed by the shell, in other words during execution the process id would be the same as the current shell which has started to execute those commands.

Once the script has been completely sourced, the shell will continue with perhaps some of the shell variables added or modified, in other words shell scripts which are supposed to be sourced are meant for the purpose of preparing an environment typically when you use a commercial compiler, then certain environment variables that are required for the compiler to work would be available in a script file and you are supposed to source that script file.


Now when you have executable script files the situation is slightly different the shell would actually launch a child shell within which that particular code will be executed, which means that the process id of the shell which is executing the shell script would be different from the parent shell which has launched it.

Once the execution is completed, then the command will be returned back to the parent shell which means that any environmental changes that are done in the executed script will be lost and usually executed scripts are meant not to change the environment but to create new functionality. So, watch out for these two intentions of the scripts before you go on to try them out.

(Refer Slide Time: 6:43)

Script location

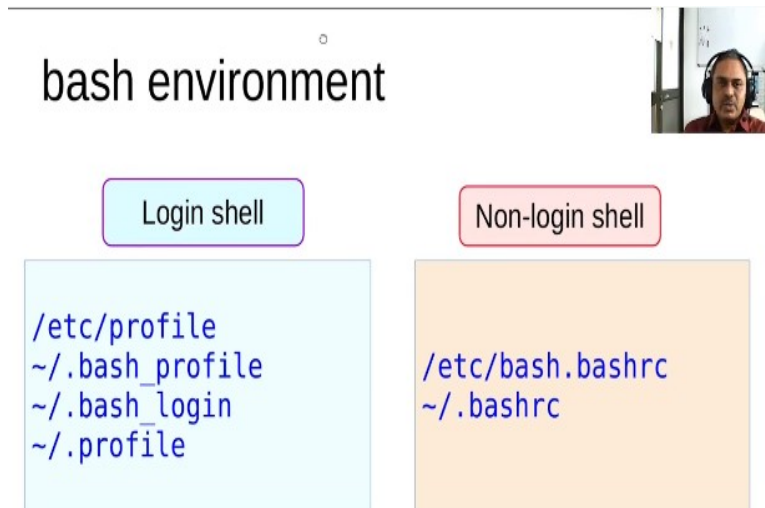
- Use absolute path or relative path while executing the script
- Keep the script in folder listed in `$PATH`
- Watch out for the sequence of directories in `$PATH`



Now where should you keep these script files? Usually when you keep them in your home directory or in any directory under the home directory, then you can run them using either absolute path or relative path and if you want to run them without mentioning the path then the script file has to be located in one of the directories which are listed in the path variable.

The sequence of directories that are listed in the path variable using the colon as a field separator will be searched for when you type a command at the command prompt and therefore, which folder you are storing the script file is important and preferably keep it in the leftmost directory so that the command that you have created would be picked up from the location where you have stored it and is not to be confused with another command of the same name elsewhere in the system.

(Refer Slide Time: 7:34)



Now to see some examples of scripts one good idea would be also to familiarize yourself with the bash environment. So, there are two broad types of shells that you enter in when you are already logged into a system and using the gnome environment if you open a terminal app, then that shell that you are opening is actually not asking you for any login and therefore it is called a non login shell.

On the other hand, when you do an SSH into a remote machine, then the shell which you are logging into would have checked for your login credentials and then opened up the command line environment for you and therefore that shell is called a login shell. Now, there are set of files that are processed to prepare the environment for you and these are different for the login shells and the non-login shells.

So, here I have listed some typical file names where such commands shell scripts are located, have a look at them to understand what are all the variables or functions that are made available to you when you log in into a bash environment either through a terminal app on your gnome user interface or when you log in using SSH to a remote computer.

Do not modify these files unless you know what you are doing, you will be able to modify these and add more functionalities by the end of the course and until then just leave them in their original form so that your shell environment is not disturbed.

(Refer Slide Time: 9:03)

Output from shell scripts



- `echo`
simple
terminates with a newline if `-n` option not given
`echo My home is $HOME`
- `printf`
supports format specifiers like in C
`printf "My home is %s\n" $HOME`

One of the very first things to do in any programming language is to type out Hello World so you would like to output something from your shell script. So, `echo` is a very simple command to output text and we have already tried that out at the command prompt and if you do not provide minus `n` option then there is the automatically a new line at the end of the text and an example of the `echo` command is given here, where the value of the shell variable dollar home will be substituted in its place and then the output would be My home is and then slash home slash username.

Now sometimes if you want to have a more fancy appearance of the output you could also use the `printf` command and it is a command and not a function but it has the same features as the equivalent of it in the C language. You could use format specifiers for string, integer, floating number etcetera just like in C language and here is one example like that. So, in `printf` commands you will have to specify backslash `n` to give a new line while printing the output.

(Refer Slide Time: 10:13)

Input to shell scripts



- `read var`

string read from command line is stored in
`$var`

Now you can also read user input into certain variables, so the command is read and you should provide the name of the variable into which the input has to be read and once it is read then the contents are accessible using the dollar var for that particular variable.

(Refer Slide Time: 10:32)

Shell Script arguments



- `$0` name of the shell program
- `$#` number of arguments passed
- `$1` or `${1}` first argument
- `${11}` eleventh argument
- `$*` or `$@` all arguments at once
- `"$*"` all argument as a **single** string
- `"$@"` all argument as **separate** strings

```
./myscript.sh -l arg2 -v arg4
```

Now there are certain inbuilt shell variables that are very useful in trying out the very first shell script that you would like to write. So, you could invoke the shell script with the different names by having symbolic links or hard links and therefore the name by which the particular script has

been invoked can be printed out using the dollar 0 and it is like asking what is my name and then the shell script would actually print out the name by which it was invoked.

Dollar hash would give you the number of arguments that are passed, so when you launch a script as given in the example myscript.sh and there are four arguments that are passed and therefore dollar hash variable would have the value 4, if you print it out from the script that is given in the bottom my script dot sh.

Dollar 1 would correspond to the very first argument that was passed, in this case of the example minus 1 would be the value of the string which corresponds to dollar 1. And if you have more than nine arguments passed, then it is a good idea to use braces to enclose the number of the argument because as you know if you would type dollar 11, then it would be interpreted as dollar 1 for the variable followed by a numeral 1 as an appended text.

It is a good habit to have braces for every shell variable while printing it out or using it in the shell environment, because as you have seen in an earlier class there are many operations that we could do to the shell variables by passing on some additional options within the braces.

Dollar star or dollar at correspond to all the arguments at once so if you want to refer to them for printing out etcetera you could do that and if you enclose the dollar star in quotes, then it would serve as a single string containing all the variables that are passed one after other using space as a delimiter and if you were to do the same thing with the dollar at within the quotes then they would also be made available but this time they are available as separate strings. So you could choose which of these two are applicable for your script and accordingly put them to use.

(Refer Slide Time: 12:52)

Command substitution



```
var=`command`
```

```
var=$(command)
```

command is executed and the output is substituted.

Here, the variable *var* will be assigned with that output.

If you wish to store the output of a command as contents of a particular variable, then you can use the command substitution feature of the bash and you could use this in two ways; one way is to use the backtick which is available just below the tilde character on the US keyboard layout. You could also use it by the feature where in single parenthesis you could enclose a command and that could also give you the output of the command as the contents of the variable.

(Refer Slide Time: 13:26)

for do loop



```
for var in list
do
    commands
done
```

commands are executed once for each item in the *list*

space is the field delimiters

set **IFS** if required

Now there are lot of loops that are available for you to think of a structured way of programming within a bash, these can be compared with the corresponding loops of the C language, very


similar in their behavior. So, there is a for do loop and for var in list, so list would be a set of words that are separated by a space, one or more spaces and the var will be assigned values one after other which are available in the list and then the commands are executed.

So, naturally the commands would use the value of the variable in those commands using the dollar and then the closing of the loop is a little bit different from languages such as C. So, for do and done would actually close the loop, sometimes if your list has the field separator different from the space then you could also inform that to the shell script by setting the variable IFS in capitals to whatever string or character that is used as a field separator.

(Refer Slide Time: 14:36)

case statement

```
case var in
    pattern1)
        commands
        ;;
    pattern2)
        commands
        ;;
esac
```



commands are executed
each *pattern* matched
for *var* in the options

There is a case statement similar to the switch case that is available in C language, it is also, it is also available in many other languages. So, you have a case statement where the variable will be compared with the patterns that are provided and wherever the matching happens then you would have the commands that will be executed.

You could also break out of some of these or provide a default option etcetera, so we will learn those when we go into the depth into the this particular case statement. Case statement is ending with the reverse of the word case, so it is ending as esac, case esac loop you can think of as the loop here.

(Refer Slide Time: 15:22)

if loop



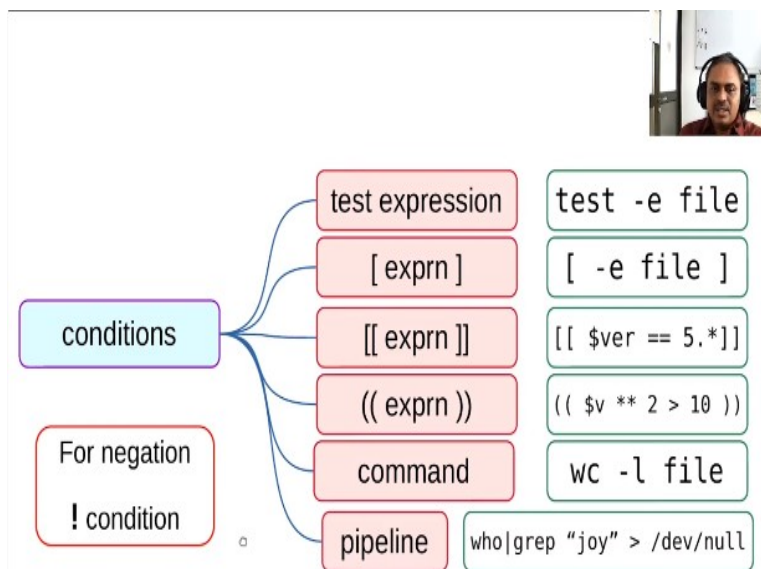
```
if condition
then
  commands
fi
```

commands are executed
only if *condition* returns
true

```
if condition; then
  commands
fi
```

The if loop would actually have conditions, test conditions which have to be evaluated to true and then the commands are executed. So, if you want the loop to look more like in a C language, then you could put a semicolon after the condition and the word then can be appearing on the same line. And the commands can be any number so you could either put them one after other using the semicolon or you could also put one below other, the loop will be closed with the letters if reversed, so it is a if fi loop that you could think of.

(Refer Slide Time: 15:58)



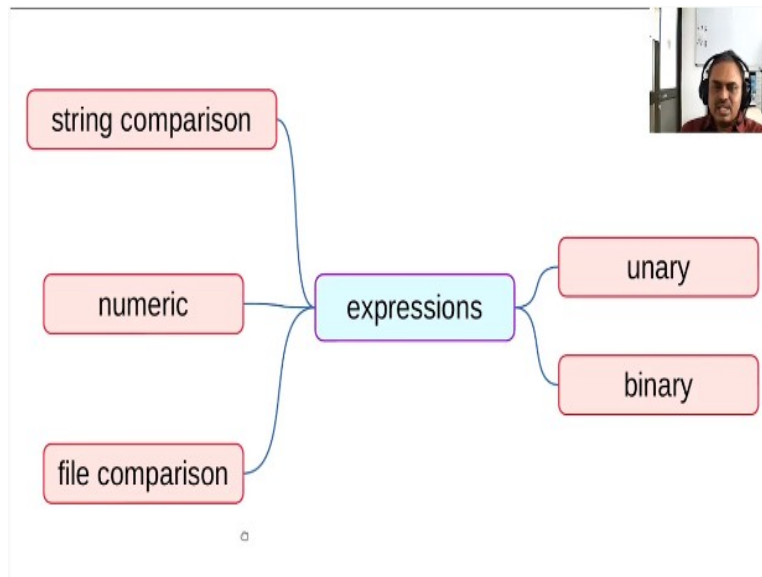
Now the conditions that are used in the if loop are very many and they are very rich in features. The simplest one would be to test the expression for the truth value by using the keyword test itself and so for example, whether the file exists or not test minus e file would give you true if the file exists in the current directory where the shell script has been launched and if it does not exist, then it would return false.

The test expressions can also be used with single square brackets. So, the same test minus e file could also be written as minus e file within the square brackets. You could use double square brackets in case you are intending to use regular expressions and other complex operations and you could put the regular expression knowledge to its full use within the test conditions by employing the double square brackets feature.

You could use the double parenthesis option also to perform complex arithmetic operations for the test conditions and you can just simply use a command and whenever the command returns true that is upon successful execution, then that means that the condition is satisfied and then you can go on to the list of commands that are within the do loop and if the command returns false, then that loop within the do and done will not be entered.

And you could also have a pipeline that is you could have a set of commands which can be combined with other commands and redirection of output and combination of logical expressions using the double ampersand or double pipe etcetera. So, all of those can also be used in place where the condition is mentioned and if you want to have a negation of that particular condition you can always use a character exclamation in front of the condition.

(Refer Slide Time: 17:55)



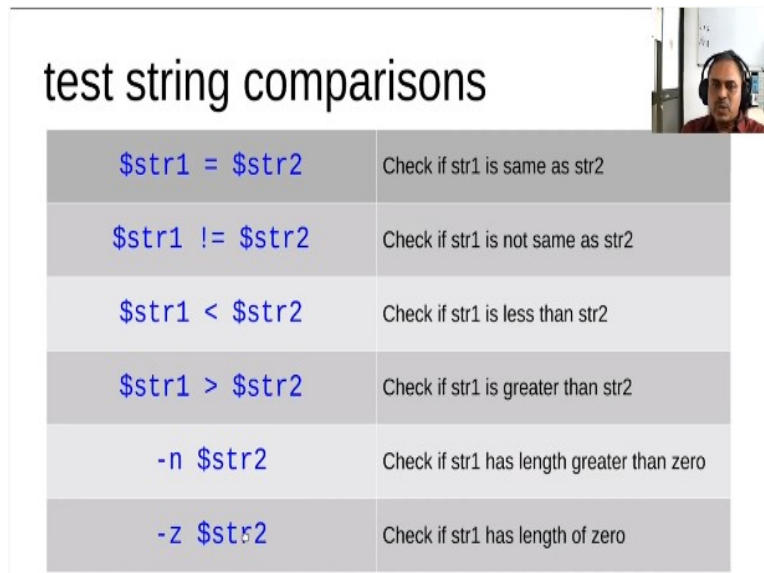
Now the expressions that we have mentioned which will come within the parentheses are also of different types, there are unary expressions and binary expressions. So, unary expression as in there will be a minus sign, there will be a hyphen followed by a single character and then there is only one argument that will be tested against. Binary expression would mean there are two arguments and then you are doing some comparison between these two arguments, and these comparisons can be either string comparison or numeric comparison or file comparison.

(Refer Slide Time: 18:23)

test numeric comparisons	
<code>\$n1 -eq \$n2</code>	Check if n1 is equal to n2
<code>\$n1 -ge \$n2</code>	Check if n1 is greater than or equal to n2
<code>\$n1 -gt \$n2</code>	Check if n1 is greater than n2
<code>\$n1 -le \$n2</code>	Check if n1 is less than or equal to n2
<code>\$n1 -lt \$n2</code>	Check if n1 is less than n2
<code>\$n1 -ne \$n2</code>	Check if n1 is not equal to n2

Now the numeric comparisons are as rich as you would have in any other programming language, so you could ask whether two numbers are equal or greater than or less than and so on.

(Refer Slide Time: 18:33)



<code>\$str1 = \$str2</code>	Check if str1 is same as str2
<code>\$str1 != \$str2</code>	Check if str1 is not same as str2
<code>\$str1 < \$str2</code>	Check if str1 is less than str2
<code>\$str1 > \$str2</code>	Check if str1 is greater than str2
<code>-n \$str2</code>	Check if str1 has length greater than zero
<code>-z \$str2</code>	Check if str1 has length of zero

And the string operations are also equally rich you could also ask whether the strings are the same or they are not the same, whether the string 1 is less than the string 2 as compared to the position of those characters within the collation sequence applicable for that particular shell and you could also ask whether the string has a non-zero length.

And this can be done in two ways minus n to ask whether the string is having a length greater than 0, so then it is true and the opposite of it using minus z whether the string has a length of 0 or not. So, you could see that minus n and minus z have opposite outcomes and we have already tried this out in an earlier exercise for the test conditions once.

(Refer Slide Time: 19:23)

Unary file comparisons


<code>-e file</code>	Check if file exists
<code>-d file</code>	Check if file exists and is a directory
<code>-f file</code>	Check if file exists and is a file
<code>-r file</code>	Check if file exists and is readable
<code>-s file</code>	Check if file exists and is not empty
<code>-w file</code>	Check if file exists and is writable
<code>-x file</code>	Check if file exists and is executable
<code>-O file</code>	Check if file exists and is owned by current user
<code>-G file</code>	Check if file exists and default group is same as that of current user

Unary file comparisons are done by using a single character after a hyphen and then the file name and you could check whether there is a file that exists by that name, whether it is a directory or a file, is it readable or is it empty, is it writable, is it executable or is it owned by the current user or the default group of that particular file, is it the same as that of the current user. So, you can perform all these checks with a single character using the unary file comparisons.

(Refer Slide Time: 19:52)

Binary file comparisons

<code>file1 -nt file2</code>	Check if file1 is newer than file2
<code>file1 -ot file2</code>	Check if file1 is older than file2



You could also have binary file comparisons to check whether a file is newer or older than another file.

(Refer Slide Time: 20:00)

while do loop

```
while condition
do
  commands
done
```

commands are executed
only if *condition* returns
true



The while loop is similar to the earlier loops that we have talked about where the condition is put to use, so whenever it is true, then the execution will enter the do loop, so do done loop with the commands in between would actually be executed only when the condition is tested to be true. You can also have an opposite feature where when the condition is false only then you need to have the loop to be executed and there you would actually have until loop.

(Refer Slide Time: 20:29)

until do loop

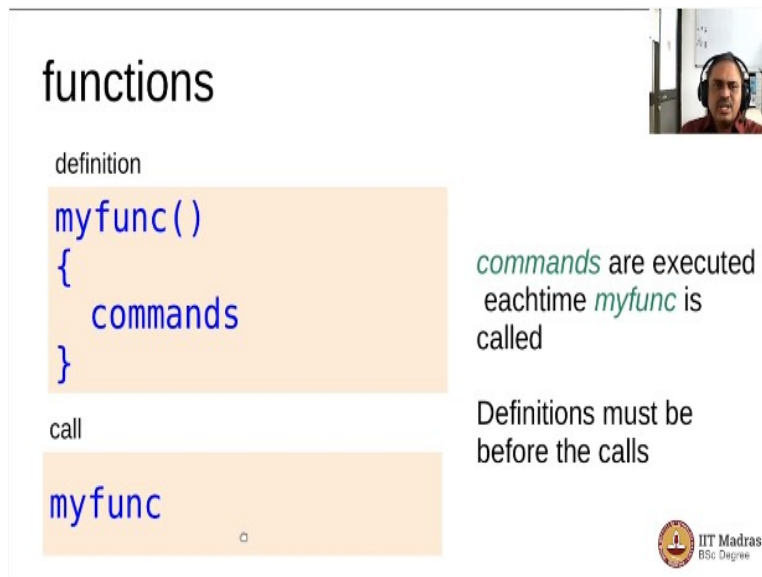
```
until condition
do
  commands
done
```

commands are executed
only if *condition* returns
false



So, until the condition is true, then the loop, then the commands will be executed which means that as soon as a condition is testing to true, then the loop execution will stop, so this is an opposite effect as while.

(Refer Slide Time: 20:42)



functions

definition

```
myfunc()  
{  
  commands  
}
```

call

```
myfunc
```

commands are executed everytime *myfunc* is called

Definitions must be before the calls

IIT Madras
BSc Degree

You could also write some functions these are not the same as the kind of functions we know in C language, it is just a collection of commands that would be executed in that sequence whenever the function is called. And calling a function is very simple just name of the function alone would be serving as a command and if you define the function before you call it, then it would understand, the shell would understand it as a function available and we could run the commands listed in the function otherwise, it would actually look up whether there is any command by that name in the operating system and if it is not existing then it would actually throw an error.

So, you could actually explore every command that you have learnt till now by combining them with the shell scripts and of course, there are also more features available in the bash programming language, we will come to them in a later session but for now we can already start putting the commands that we have learned till now to a good use by writing those commands within a script file making it executable and then we have our scripts ready, thanks to the command line editors we can already create those scripts and test them out right inside the terminal itself.