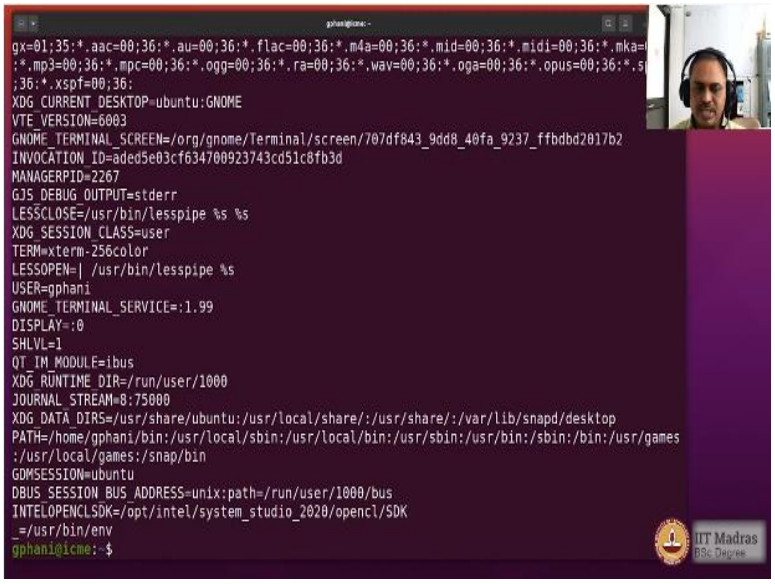
System Commands
Professor Gandham Phanikumar
Department of Metallurgical and Materials Engineering
Indian Institute of Technology, Madras
Lecture 10
Shell Variables – Part 2

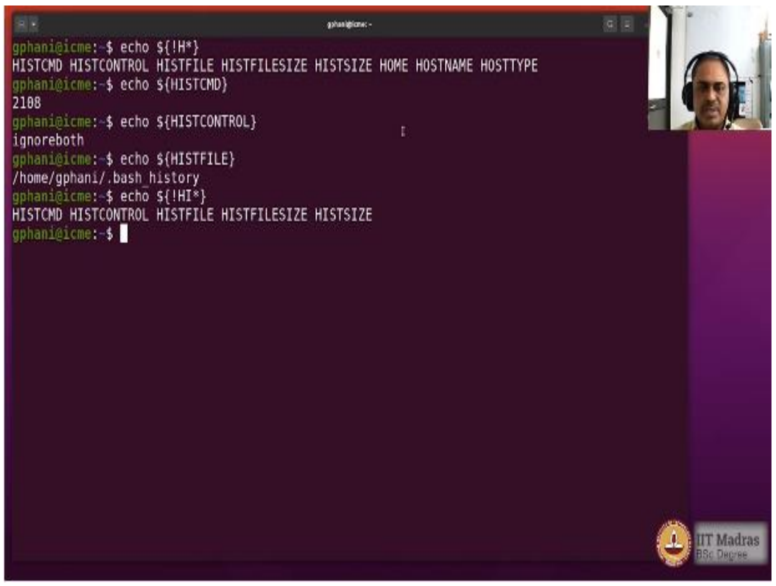
(Refer Slide Time: 0:15)



```
gx=01;35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;36:*.mka=
:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.s
;36:*.xspf=00;36:
XDG_CURRENT_DESKTOP=ubuntu:GNOME
VTE_VERSION=6003
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/707df843_9dd8_40fa_9237_ffbdbd2017b2
INVOCATION_ID=aded5e03cf634700923743cd51c8fb3d
MANAGERPID=2267
GJS_DEBUG_OUTPUT=stderr
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
LESSOPEN=| /usr/bin/lesspipe %s
USER=gphani
GNOME_TERMINAL_SERVICE=:1.99
DISPLAY=:0
SHLVL=1
QT_IM_MODULE=ibus
XDG_RUNTIME_DIR=/run/user/1000
JOURNAL_STREAM=8:75000
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share/:/usr/share/:/var/lib/snapd/desktop
PATH=/home/gphani/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
:/usr/local/games:/snap/bin
GDMSESSION=ubuntu
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
INTELOPENCLSDK=/opt/intel/system_studio_2020/opencl/SDK
_=/usr/bin/env
gphani@icme:~$
```

Let us say we would like to inspect what are all the variables that are present within the shell environment. We have already seen that in the past so you could use a command like printenv, and you will see the whole bunch of stuff that is displayed on the screen and you will see that there is a lot of variables that are available which are now listed for us, you could also use the command env and that will also display some of these.

(Refer Slide Time: 0:42)

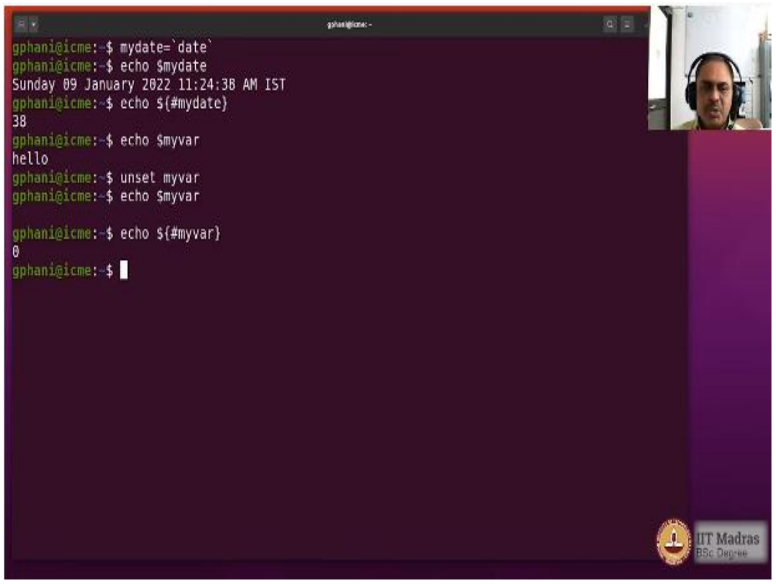


```
gphani@icme:~$ echo ${!H*}
HISTCMD HISTCONTROL HISTFILE HISTFILESIZE HISTSIZE HOME HOSTNAME HOSTTYPE
gphani@icme:~$ echo ${HISTCMD}
2108
gphani@icme:~$ echo ${HISTCONTROL}
ignoreboth
gphani@icme:~$ echo ${HISTFILE}
/home/gphani/.bash_history
gphani@icme:~$ echo ${!HI*}
HISTCMD HISTCONTROL HISTFILE HISTFILESIZE HISTSIZE
gphani@icme:~$ 
```

Now, there is a slightly better way of going about inspecting what variables are present in the memory, that is by using a particular command as follows echo and here what we will do is you put the exclamation mark followed by any character, so the character starting with let us say so, this is a construction to see what are the names of the variables, it starts with H and you see that these are the variables and each of these now you can actually explore, the values.

So, each of them can be done and so on, so this is a very interesting command because it is actually giving you the names of the variables rather than the value of the variable, the star actually is an expansion to tell you that the name can be anything after H, so of course, you can also start with the multiple characters, so you could actually ask like this echo dollar bang, which means give me the names of the variables, not the value and let us say HI followed by star which means that all those starts with the HI will then be listed, you see that the home is not listed, because it is HO. So, this is a very nice way because if you want to inspect what are the variables that you could actually check it out.

(Refer Slide Time: 2:17)



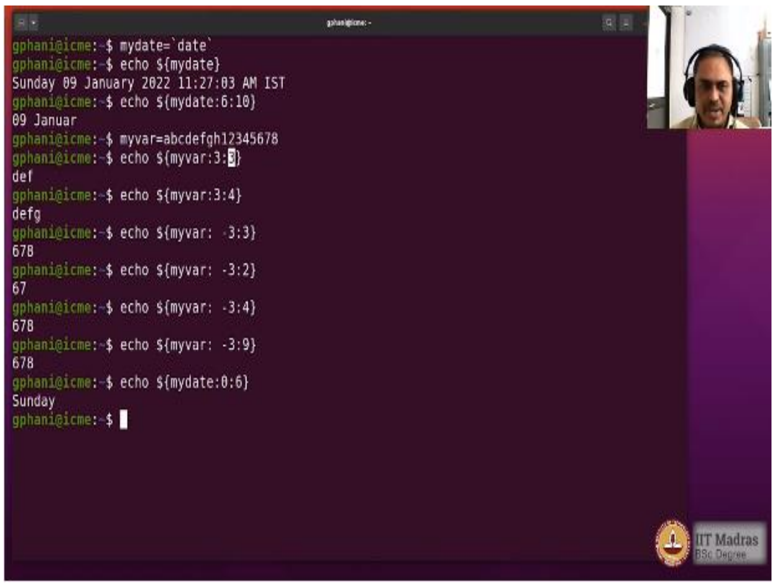
```
gphani@icme:~$ mydate=`date`
gphani@icme:~$ echo $mydate
Sunday 09 January 2022 11:24:38 AM IST
gphani@icme:~$ echo ${#mydate}
38
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ unset myvar
gphani@icme:~$ echo $myvar

gphani@icme:~$ echo ${#myvar}
0
gphani@icme:~$
```

Now, let us inspect some more features that are available. So, mydate is a string in a variable called mydate, I would like to store the output from the command date and you see that mydate is now storing a particular string. So, I do not want to type such a long string, so I want to use this shortcut to actually also illustrate certain features. Now, let us say I want to count how many characters are there in this particular string. So, you see that the hash at the very beginning is indicating that it should not print the value of the particular variable, but the length of the particular variable.

So, the length happens to be 38 characters, so this is 38 characters, you could count it off. So, as you can see, the braces are actually very powerful, because the very first character tells you what to do with that. So, the first character is bang then it actually asks for the names of the variables if it is hash, then it is asking for the length of the value of the particular variable.

Now, let us say if the variable is not set, what would come out. So, let us see myvar something is set, so unset myvar, now, myvar is not set, so if I now ask for the length of that particular variable, and now you see that it says 0, because it is not existing, so it has 0 length. Now, the colon within the braces is also very powerful as we have seen earlier, where we can take action to display something that is useful to give us a default value, whether the variable is present or absent. There are two different ways to go about that.

(Refer Slide Time: 4:18)



```
gphani@icme:~$ mydate=`date`
gphani@icme:~$ echo ${mydate}
Sunday 09 January 2022 11:27:03 AM IST
gphani@icme:~$ echo ${mydate:6:10}
09 Januar
gphani@icme:~$ myvar=abcdefgh12345678
gphani@icme:~$ echo ${myvar:3:3}
def
gphani@icme:~$ echo ${myvar:3:4}
defg
gphani@icme:~$ echo ${myvar: -3:3}
678
gphani@icme:~$ echo ${myvar: -3:2}
67
gphani@icme:~$ echo ${myvar: -3:4}
678
gphani@icme:~$ echo ${myvar: -3:9}
678
gphani@icme:~$ echo ${mydate:0:6}
Sunday
gphani@icme:~$ 
```

Now, we will see some more uses of that particular colon. Now, I would like to illustrate how to extract a part of a string from the value of a particular variable. So, for that, we take the mydate as the variable and look at the value of the variable and we like to extract let us say the characters in a particular range. So, you could do that in this fashion, so from let us say sixth character onwards, I would like to start and go up to it let us say 16, so ten characters I would like to display after that, and you see that it is done here, so after 6 characters, it starts to basically take up to 10.

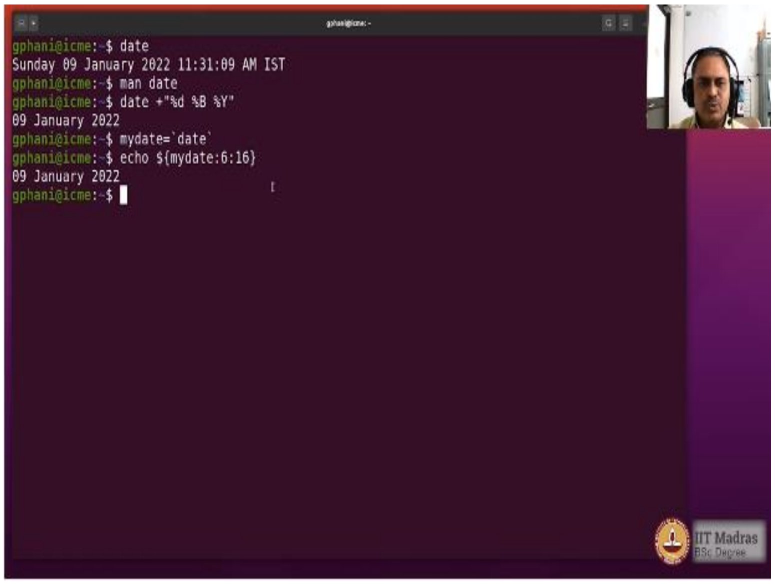
Now, this is illustrated quite nicely if you have a string in which you can actually count better. So, let us do that here, so myvar is equal to and let us say abcdefgh12345678, so now you will be able to relate to it quite easily. So, echo dollar myvar and colon and now let us say I put three colon 3 now, you see that the first three are skipped following that the three characters are being displayed.

Now, I would make this as four and you see there are four characters that are displayed. So, which means that this part is to indicate what is the offset and this part is to indicate how many characters have to be displayed from the offset. Now, you could also have the offset negative that is offset from the right-hand side. However, you know that minus is used for a specific purpose already it is used to inspect whether the variable was present or not. So, a minus sign is to be always preceded with a blank to indicate not to be confused with the other operation.

So, let us do that here minus 3 colon 3, so now you see that it is actually coming from the right-hand side, so it has offset the characters by three from the right-hand side and printed three of them. Now, if I print only two of them, you will see that only 6 and 7 are displayed 8 is not displayed. Now, if I happen to ask for more characters than they are in the string, then it would not complain, it would print the how many are available. So, you could also have them as let us say 8 or 9 or something, does not matter, only three are available after the offset and therefore those manually are displayed.

So, you can see that if you have a string, you can actually print portions of the string either from the left or from the right by taking an offset and also indicating the number of characters after the offset. So, if I want to extract from the date command only the day then what do we do, echo dollar mydate colon and we would have no offset and the six characters and you see that the day of the week has been displayed. So, this is a very neat way by which you can extract portions of the strings from the value of a variable which is already coming from the output of the command.

(Refer Slide Time: 7:45)



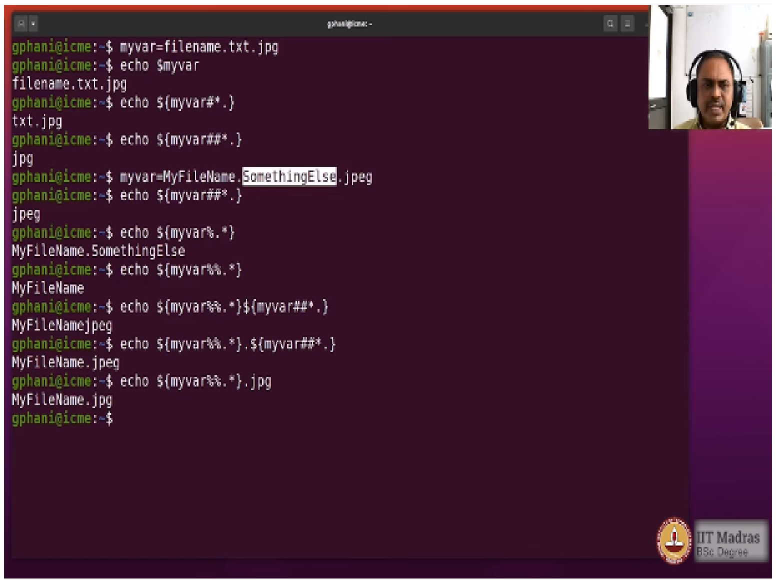
```
gphani@icme:~$ date
Sunday 09 January 2022 11:31:09 AM IST
gphani@icme:~$ man date
gphani@icme:~$ date +"%d %B %Y"
09 January 2022
gphani@icme:~$ mydate=`date`
gphani@icme:~$ echo ${mydate:6:16}
09 January 2022
gphani@icme:~$ 
```

Let us say we would like to extract only the date from the date command. So, it can be done in two ways, so I would like to only extract this part, let us say only that part can be done in two ways. So, first, I would illustrate that by using the date command itself, so look at the manpage for date command and you can see that certain fields can be displayed, so you can see that the percentd can be used for displaying the date of the month, then percent B for the full month name and the percent capital Y for the year, in four digits. So, we could do this by the date command, so date plus and then the format so percent d and then a blank space percent capital B then blank space percent capital Y. So, you can see that we have printed the date by looking at only particular set of fields.

Now, we could also extract the fields from the output which is already available for whatever reason, if you want to do that, you could also do it in this manner mydate is equal to, so now I have stored the string within the variable mydate and echo mydate colon, then I would have 6 colon 16 and you see that output is same, so which means that you could also cut portions of your string and print them by using the manipulation of the value of a variable by using the colon.

So, what comes after the colon the first number is the offset the second number is the length of the string after the offset. And that gives you basically the same output as what we have achieved by using a particular command option for the date command. So, sometimes the options do not have the possibilities for the manipulation that we want and therefore learning other ways of manipulating the string is also useful.

(Refer Slide Time: 9:54)



```
gphani@icme:~$ myvar=filename.txt.jpg
gphani@icme:~$ echo $myvar
filename.txt.jpg
gphani@icme:~$ echo ${myvar#*.}
txt.jpg
gphani@icme:~$ echo ${myvar##*.}
jpg
gphani@icme:~$ myvar=MyFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar##*.}
jpeg
gphani@icme:~$ echo ${myvar%.*}
MyFileName.SomethingElse
gphani@icme:~$ echo ${myvar%%.*}
MyFileName
gphani@icme:~$ echo ${myvar%%.*}${myvar##*.}
MyFileNamejpeg
gphani@icme:~$ echo ${myvar%%.*}.${myvar##*.}
MyFileName.jpeg
gphani@icme:~$ echo ${myvar%%.*}.jpg
MyFileName.jpg
gphani@icme:~$
```

Now, I would like to illustrate the concept of extracting patterns out of a string. We will be learning about the regular expressions and the grip shortly but we can already get introduced to that concept by trying out some patterns, very simple patterns, which are very useful to inspect the file extensions. So, I would have let us say myvar as file name dot txt dot jpg. So, it is a string which happens to have two extensions for whatever reason. So, that is a variable myvar in which the particular string with the two extensions are there.

Now, I would like to extract only the last extension by a particular pattern. So, what is the pattern? The pattern is it is following after the dot, so the dot is used as a placeholder and then we can use star for unknown number of characters to be matched and let us see what we can do with respect to the pattern matching. So, there are two ways of pattern matching, whether we would like to the match pattern to be displayed or the matched pattern to be hidden, so both are available.

So, here we have written in a particular manner, you can see that there is a hash which is coming after the variable name, we have already seen hash coming up, so if it came in front of the variable, that is number of characters are accounted for particular variable, so it is coming after the name of the variable and then there is a pattern, the pattern is star dot, so you see what happens when you try to echo and you see that a pattern that is matching star dot is displayed.

Now, if you have one hash it would actually see what is the minimum that you can match with the star, so when the file name is matched and then after that just stopped, but technically file name dot txt itself can be matched as a star dot because there is one more dot available, so that can be achieved by asking it to maximize the amount of matching by using double hashes. So, this idea of double hash or double percentage sign etcetera that we will see are used to indicate a minimum and maximum matching. So, you see that the extension has come out now, this extension is not by counting number of characters, but by using the pattern where a dot is used as a separator.

So, how do we verify, let us do this way. So, myvar is equal to let us say myfilename dot somethingelse dot and then jpeg now, the extension is four characters. And now, if you see here jpeg is coming out which means that we are not going by the number of characters that are being matched, but by a pattern and the pattern is star dot, which means that after an unknown number of characters so there should be a dot and that should be matched as largely as possible by using the double hash symbols.

So, the entire myfilename dot somethingelse dot will be matched for the pattern that is being displayed here and then the remaining part is actually displayed as the output so jpeg is coming out as an output. So, this is quite useful to find out what is the extension or that particular file name that is coming into the script by ignoring whatever is in front of the dot symbol.

Now, the opposite of it is to display what has not been matched, so that is used done using the percentage symbol, percentage and now, I would like to match what is in front and therefore, I would use the dot star as a way to match, so dot star and you see that whatever is in front of the dot has now been matched and it is matching as minimal as possible, because only this part is been removed.

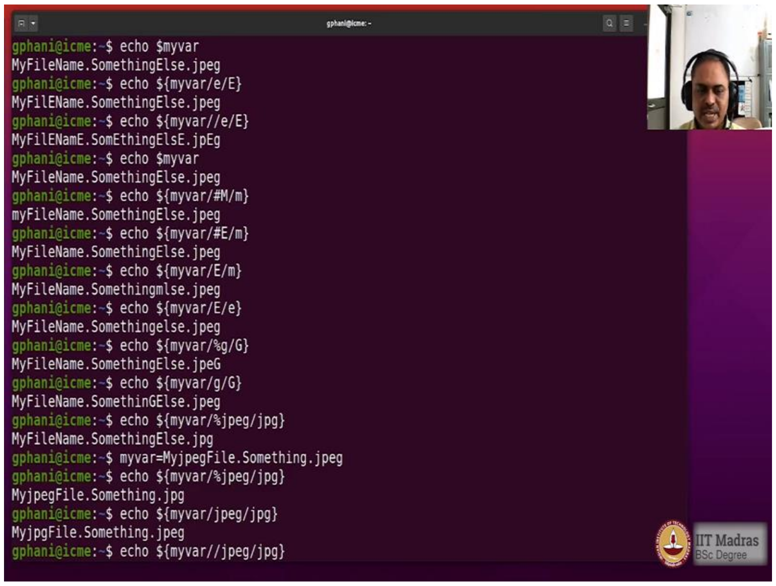
Now, I would like to remove this part alone fully so that only the stem of the file name is shown, so for that as I mentioned earlier double usage of that particular trigger can be useful, so we use the symbol twice here, then the maximum possible matching will be done and now, you can see the only the stem is shown myfilename, so dot somethingelse dot jpg has been matched with the pattern that we have put as dot star and then the remaining part is now displayed. So, now, you see that the use of hash as well as a percentage sign, percentage sign is showing you what is in front of the dot and then the hash is showing you what is after the dot by the way we have given the patterns.

So, now let us say I would like to make the name of this particular variable to be such that it is only the stem followed by the extension. So, we could do that in this manner echo dollar myvar percentage percentage dot star that will give me the stem and then dollar myvar hash hash star dot that will give me only the extension.

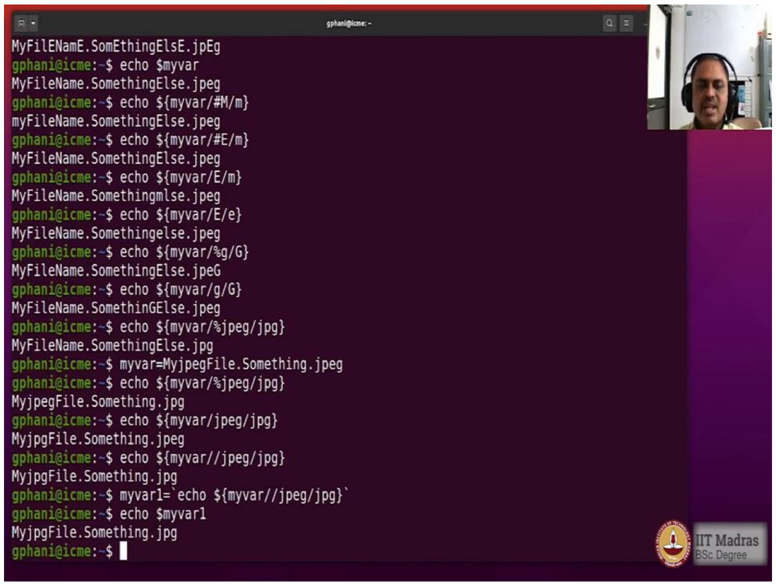
And you see that we have got it as filename jpg, and now, I would like to have a dot in between them, so that its like a file name dot extension and I have got that now modified. So, you can see that we have been able to manipulate a string myfilename dot somethingelse dot jpg as the stem dot the extension alone, without even worrying about how much of the text is in between the two dots, and we have just knocked it off and created this of course, we can also use it to give the extension in a different manner also.

So, for example, you could actually make it like this and so depending upon our requirements of the script, these kinds of manipulations can be quite useful and these are also very fast because these are inbuilt features for the bash.

(Refer Slide Time: 16:06)



```
gphani@icme:~$ echo $myvar
MyFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/e/E}
MyFilEName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar//e/E}
MyFilENamE.SomEthingElsE.jpEg
gphani@icme:~$ echo $myvar
MyFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/#M/m}
myFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/#E/m}
MyFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/E/m}
MyFileName.Somethingmlse.jpeg
gphani@icme:~$ echo ${myvar/E/e}
MyFileName.Somethingelse.jpeg
gphani@icme:~$ echo ${myvar/%g/G}
MyFileName.SomethingElse.jpeG
gphani@icme:~$ echo ${myvar/g/G}
MyFileName.SomethinGElse.jpeg
gphani@icme:~$ echo ${myvar/%jpeg/jpg}
MyFileName.SomethingElse.jpg
gphani@icme:~$ myvar=MyjpegFile.Something.jpeg
gphani@icme:~$ echo ${myvar/%jpeg/jpg}
MyjpegFile.Something.jpg
gphani@icme:~$ echo ${myvar/jpeg/jpg}
MyjpgFile.Something.jpeg
gphani@icme:~$ echo ${myvar//jpeg/jpg}
```



```
MyFilENamE.SomEthingElsE.jpEg
gphani@icme:~$ echo $myvar
MyFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/#M/m}
myFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/#E/m}
MyFileName.SomethingElse.jpeg
gphani@icme:~$ echo ${myvar/E/m}
MyFileName.Somethingmlse.jpeg
gphani@icme:~$ echo ${myvar/E/e}
MyFileName.Somethingelse.jpeg
gphani@icme:~$ echo ${myvar/%g/G}
MyFileName.SomethingElse.jpeG
gphani@icme:~$ echo ${myvar/g/G}
MyFileName.SomethinGElse.jpeg
gphani@icme:~$ echo ${myvar/%jpeg/jpg}
MyFileName.SomethingElse.jpg
gphani@icme:~$ myvar=MyjpegFile.Something.jpeg
gphani@icme:~$ echo ${myvar/%jpeg/jpg}
MyjpegFile.Something.jpg
gphani@icme:~$ echo ${myvar/jpeg/jpg}
MyjpgFile.Something.jpeg
gphani@icme:~$ echo ${myvar//jpeg/jpg}
MyjpgFile.Something.jpg
gphani@icme:~$ myvar1=`echo ${myvar//jpeg/jpg}`
gphani@icme:~$ echo $myvar1
MyjpgFile.Something.jpg
gphani@icme:~$ 
```

Now, I will illustrate how to do a pattern matching, which is to also do the replacement. So, earlier what we have done is to display what has been matched or to display what has not been matched by using the symbol hash and percentage. Now, we can also do a replacement of whatever has been matched.

So, here we would actually illustrate that and there are four different ways of doing that. So, what I would do is echo myvar, now pattern matching in most of the Linux environment always goes with a pair of forward slashes, so that is something that we will use in GREP when we learn about regular expressions later on, but already we can start trying it out. So, if you use it once, it will actually replace it once, so what I would like to do is whenever I have got small e I would like to make it as a capital E let us say, small e I like to make it as capital E.

So, you see that the display is done for the whole variable and the first e is been replaced with the capital E but the second e has not been done and the third one also has not been touched, so which means that when you use this forward slash once it replaces only once.

Now, if you would like it to replace as many times as it occurs, so you will put twice of that and you see that every time the small e has occurred it has been replaced with capital E, so you could see that everywhere it has been done and this difference is coming because you have used this particular pattern matching trigger twice, so normally the same convention is applicable, so if we have used the trigger once it is done once and if you have done it twice it means it will done it will do the matching the maximum number of times that is possible. So, it has been observed for all the implementations of string manipulations as we have seen till now.

Sometimes you may want to check whether this pattern is matching should be done in the beginning of the string or at the end of the string. So, let us do that out, let us just try that out. So, let us say if the file name starts with the capital M then you should change it with small m, so we could try that out.

Here the pattern is actually preceded with a hash and let us say a small m, capital M, so you see that the beginning M has been changed to small m and nothing else has occurred. Now, we could also check this out with respect to E. So, you see that the capital E that is there, is the first capital E within this particular string, but it has not been changed because the hash in front of capital E indicates that you should change it only if it occurs in the beginning of the string and therefore, this will not affect, so what is the difference if I were to actually not have the hash then the first occurrence of capital E will be changed with the m and you see that here it has been done and this is not what we intended.

So, something like that you could actually trigger whether it has to be matched in the beginning or somewhere in the middle. So, I will correct it with e so that it actually also makes some sense in English. Now, you could also ask the same thing to be done at the end of the string, so echo dollar, myvar and the past pattern I wanted to the end so, I would put it as a percentage symbol and if the end of the string happens to be small g I would like to make it with a capital G.

Now, you see that only the end of the string has been modified to capital G, but there is one more g here in the middle that has not been touched. Now, what will happen if you did not have the percentage, you see that whatever is in the middle has been encountered first while parsing, so parsing is generally done from the beginning of the string and therefore, this G is the first one, so that has been made into capital and the last g of the string has not been touched.

So, you see the difference between this usage and this usage, so if you have preceded the pattern to be matched with a percentage sign symbol, then it will be matching with the end of the string, but if you did not have it, it will be done at the first occurrence of the string.

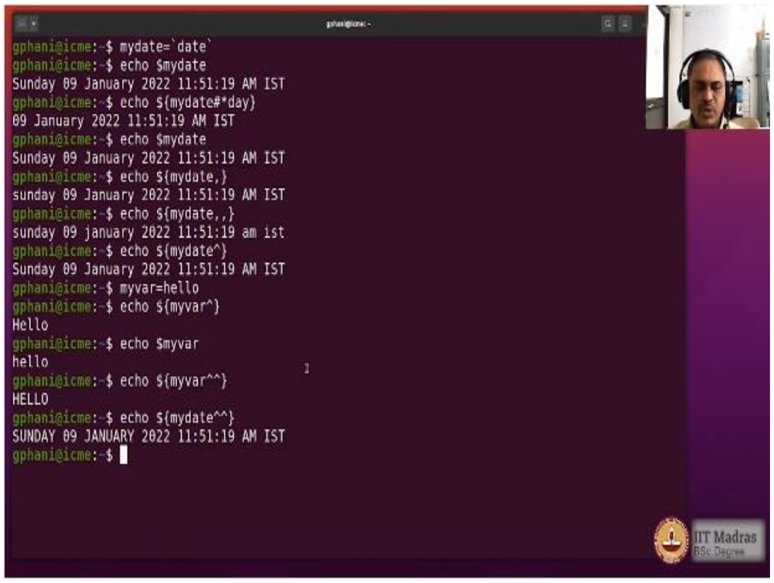
There are four different ways of matching that is basically once or many times or at the beginning of the string or at the end of the string. Now, it’s up to you to imagine how you can combine these to ensure that you are basically are able to modify the names as you like. So, in this case, for example, I want to just run a script which changes the string jpeg to jpg, but I want it only if it is in the end of the string so that it is only affecting the file extension and not the name of the file.

So, what I would do here is as follows, jpg and I would like it to be only in the end, so I put a percentage symbol there and you see that the file extension alone has been tampered with and everything else is present. So, how do we verify that, let us say I will change the name of the variable to be like this myvar is equal to and I say myjpegfile dot something dot jpeg. Now, you see that I have got two occurrences of jpeg as a string and you see here, the first occurrence of jpeg is not touched, the last occurrence only has been touched and that has been changed to jpg because of the percentage symbol.

Now, I would try what happens if I did not do that, so you can see that here the first occurrence of jpeg has been changed to jpg, but the last occurrence has been ignored. Now, I want to change everywhere for example, I would obviously have the possibility to use a forward slash twice and it will do everywhere that is possible. So, you can see jpeg is occurring twice and in the beginning of the string as well as at the end of the string, and both those portions have been changed to jpg.

So, you can see that you can actually match portions of the string and replace them or extract or delete them. So, you have all these possibilities, now, by a combination of all these you can actually go on to try what you like with respect to those variables, how do we actually store the output of one operation in another variable, so that we can then go on to do further operations, we already have the back quote for that. So, we will do like this myvar1 is equal to back quote eco, then we can actually have this myvar, now echo myvar1 and you see that now the myvar1 is contained in the output of whatever manipulation we have done.

(Refer Slide Time: 23:47)



```
gphani@icme:~$ mydate=`date`
gphani@icme:~$ echo $mydate
Sunday 09 January 2022 11:51:19 AM IST
gphani@icme:~$ echo ${mydate#*day}
09 January 2022 11:51:19 AM IST
gphani@icme:~$ echo $mydate
Sunday 09 January 2022 11:51:19 AM IST
gphani@icme:~$ echo ${mydate,}
sunday 09 January 2022 11:51:19 AM IST
gphani@icme:~$ echo ${mydate,,}
sunday 09 january 2022 11:51:19 am ist
gphani@icme:~$ echo ${mydate^}
Sunday 09 January 2022 11:51:19 AM IST
gphani@icme:~$ myvar=hello
gphani@icme:~$ echo ${myvar^}
Hello
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ echo ${myvar^^}
HELLO
gphani@icme:~$ echo ${mydate^^}
SUNDAY 09 JANUARY 2022 11:51:19 AM IST
gphani@icme:~$
```

Now, we can actually illustrate this by extracting the day out of the date string, so mydate is equal to back quote date, so echo mydate will contain the string, now I would like to just manipulate what is the first word Sunday, so mydate, then I have got hash star day so that would match the Sunday and there is nothing else that is matching with the day there.

And you see that the Sunday alone has been knocked off and rest of it is displayed. So, you could actually try these kinds of manipulations on strings, which are outputs of other commands, so that you can achieve the kind of pattern that you are looking for.

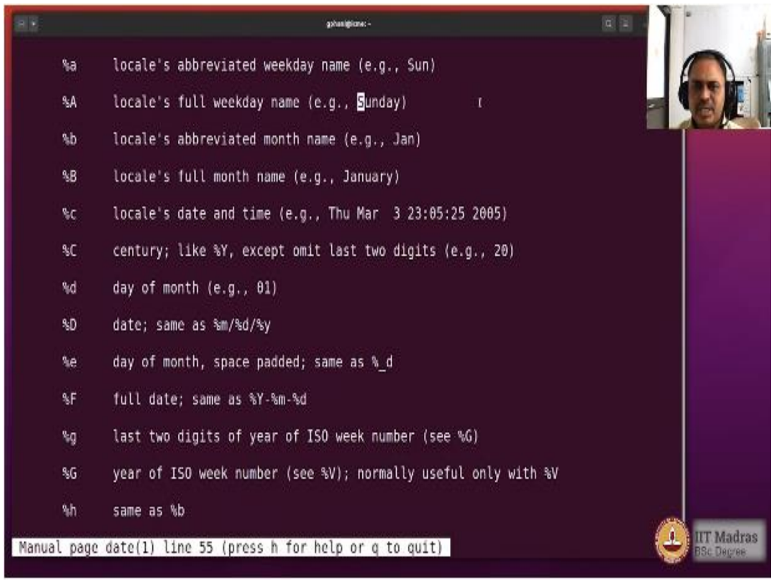
Now, let us say we would like to also do some manipulations with respect to the case change, that is the uppercase to lowercase and vice versa. So, those are also available and they are very powerful because we will see that the speed of those operations is much faster than if you were to do by a separate program.

So, let us do that here mydate is with the combination of capital letters as well as small letters. So, I would now do a manipulation, so with a comma so I just use a comma there and you see that only the first character has been made to a small letter the January J is still a capital letter and if I want the entire string to be in a small letter, so though I put two commas and you see that the January also has a small j, ist also has a smaller ist, AM also is changed to small am so, comma or two commas basically indicate whether you would like to have the first character to be changed to a lower case or all the characters to be changed to the lowercase.

The same thing can be applied in a reverse manner for uppercase where instead of comma you would actually have a hat, hat now this is not going to be very useful, because Sunday for the original string already has a capital letter, so for that, I would use some other variable, so myvar is equal to hello and echo myvar hat. So, you see that the H is now capital, so original variable it does not have a capital H in the beginning, now it is capital H if I use hat.

Now, if I use double hat what happens? So, by the same connotation, all the letters will be changed to uppercase and you can see that has been done. So, this can be also applicable for the string that we have just now tried mydate double hat and you see the entire date string has been converted to uppercase. So, this is quite useful in some pattern matching in situations where you need to work with full capital letters or small letters as the convention for that particular file format may require and you can do those conversions within no time just by adding these trailing characters after the variable name within the braces. So, you can see that this construction of a variable with the braces is very powerful, because by having things within the braces, there are commands that can actually operate on the value of that particular variable.

(Refer Slide Time: 27:25)

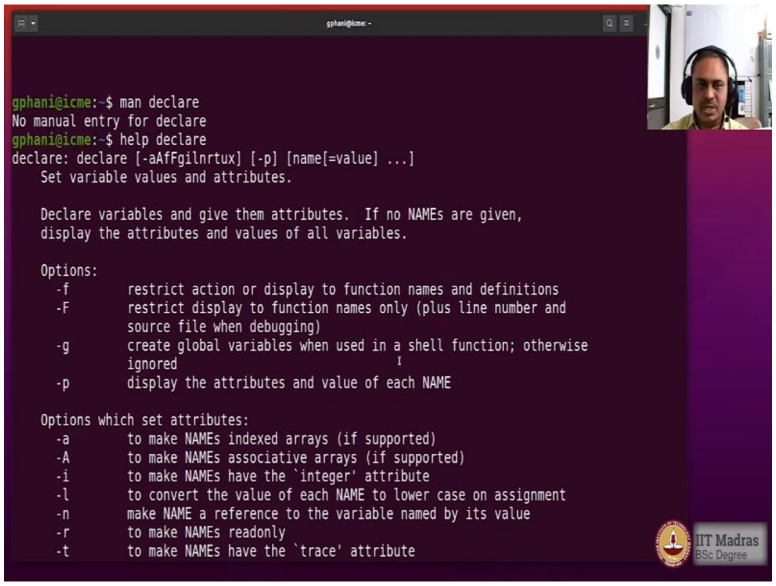


```
       %a     locale's abbreviated weekday name (e.g., Sun)

       %A     locale's full weekday name (e.g., Sunday)

       %b     locale's abbreviated month name (e.g., Jan)

       %B     locale's full month name (e.g., January)

       %c     locale's date and time (e.g., Thu Mar  3 23:05:25 2005)

       %C     century; like %Y, except omit last two digits (e.g., 20)

       %d     day of month (e.g., 01)

       %D     date; same as %m/%d/%y

       %e     day of month, space padded; same as %_d

       %F     full date; same as %Y-%m-%d

       %g     last two digits of year of ISO week number (see %G)

       %G     year of ISO week number (see %V); normally useful only with %V

       %h     same as %b
 Manual page date(1) line 55 (press h for help or q to quit)
```

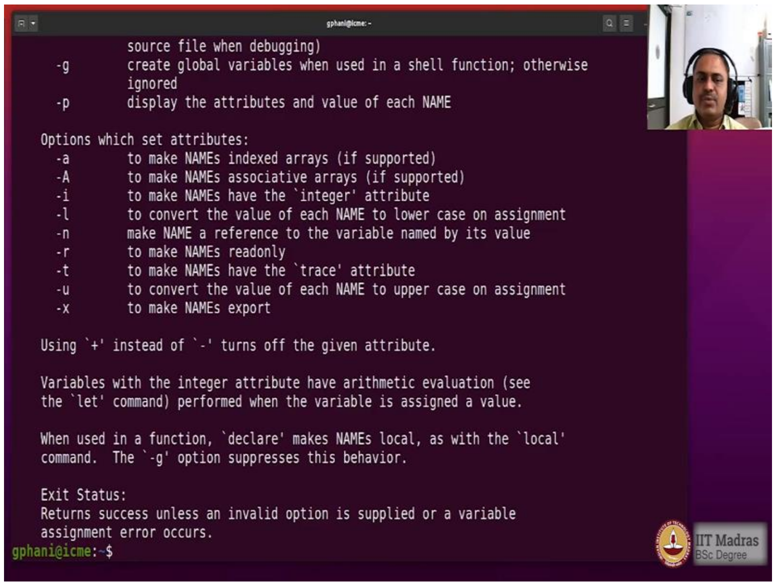
Now, you can look at the manpage for date and you see that there is actually no easy manner by which I can actually have the day of the week in a capital letter. So, you can see that capital A is for the week day, but it is such that the first character is capital rest of it is small. So, if I want Sunday to be displayed as full caps, then it is not readily available and these are the kinds of situations where here you have got ability to do it by just extracting the string and then manipulating the string.

You have seen that we have been assigning values to variables of different types and we are also changing the type of value for the same variable. So, sometimes it is alphanumeric, sometimes it says a number and we were also able to change the case of the string uppercase characters or lowercase characters as we would like to have them as, so is it possible to have the restrictions on the variable so that the values can be either only uppercase characters or only lowercase characters, only a string type of a value or only numerical value, such a restriction is possible on the variables using the declare command and let us explore that now to see how to restrict the values that can be assigned to a particular shell variable.

(Refer Slide Time: 28:46)



```
gphani@icme:~$ man declare
No manual entry for declare
gphani@icme:~$ help declare
declare: declare [-aAfFgilnrtux] [-p] [name[=value] ...]
    Set variable values and attributes.

    Declare variables and give them attributes.  If no NAMEs are given,
    display the attributes and values of all variables.

    Options:
      -f        restrict action or display to function names and definitions
      -F        restrict display to function names only (plus line number and
                source file when debugging)
      -g        create global variables when used in a shell function; otherwise
                ignored
      -p        display the attributes and value of each NAME

    Options which set attributes:
      -a        to make NAMEs indexed arrays (if supported)
      -A        to make NAMEs associative arrays (if supported)
      -i        to make NAMEs have the `integer' attribute
      -l        to convert the value of each NAME to lower case on assignment
      -n        make NAME a reference to the variable named by its value
      -r        to make NAMEs readonly
      -t        to make NAMEs have the `trace' attribute
```

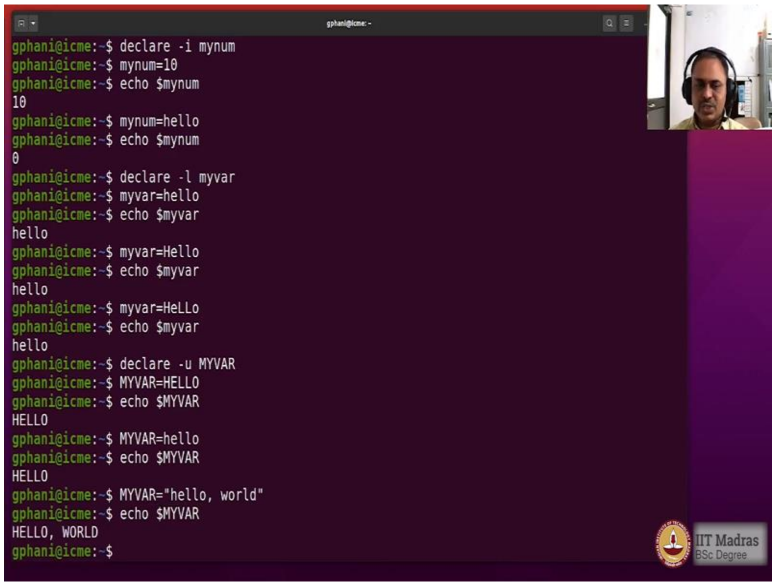


```
                source file when debugging)
      -g        create global variables when used in a shell function; otherwise
                ignored
      -p        display the attributes and value of each NAME

    Options which set attributes:
      -a        to make NAMEs indexed arrays (if supported)
      -A        to make NAMEs associative arrays (if supported)
      -i        to make NAMEs have the `integer' attribute
      -l        to convert the value of each NAME to lower case on assignment
      -n        make NAME a reference to the variable named by its value
      -r        to make NAMEs readonly
      -t        to make NAMEs have the `trace' attribute
      -u        to convert the value of each NAME to upper case on assignment
      -x        to make NAMEs export

    Using `+' instead of `-' turns off the given attribute.

    Variables with the integer attribute have arithmetic evaluation (see
    the `let' command) performed when the variable is assigned a value.

    When used in a function, `declare' makes NAMEs local, as with the `local'
    command.  The `-g' option suppresses this behavior.

    Exit Status:
    Returns success unless an invalid option is supplied or a variable
    assignment error occurs.
gphani@icme:~$
```

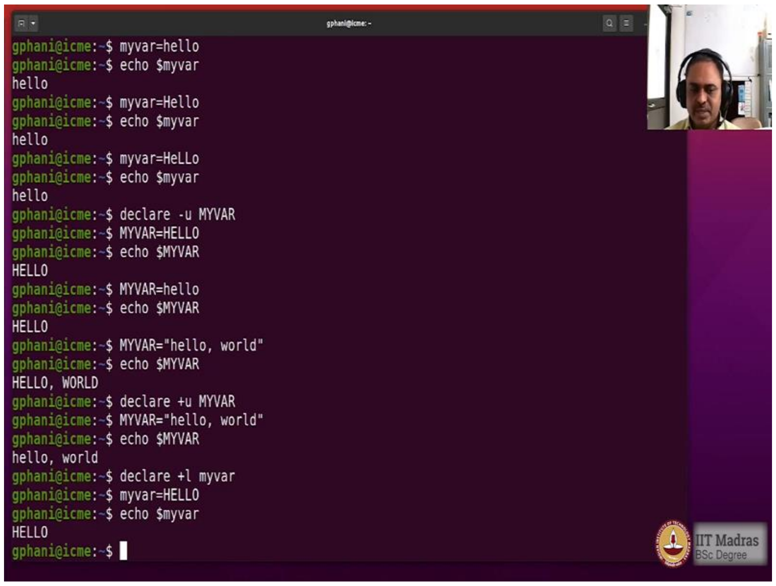
Now, let us look at the help for the declare command and you see that it is not there, so which means that it is not a manpage. So, it may be a shell bulletin and therefore now you see that it is available. So, it is actually coming from the shell, so what type of a command is that? So, it says it is a shell bulletin, so help declare what are the various features that are available.

Now, you can see that there are some options that are there, which go in front of the name of the variable. So, here for example, minus a says that the particular variable is actually an indexed array, or it can be an associative array, or it can be an integer, uppercase characters, or it can have lowercase characters and so on. And you can actually use the plus sign to unset that particular restriction and minus sign to actually set it. That is a little counterintuitive, but normally all options come with a minus sign, so therefore, it is used in that sense.

(Refer Slide Time: 30:03)



```
gphani@icme:~$ declare -i mynum
gphani@icme:~$ mynum=10
gphani@icme:~$ echo $mynum
10
gphani@icme:~$ mynum=hello
gphani@icme:~$ echo $mynum
0
gphani@icme:~$ declare -l myvar
gphani@icme:~$ myvar=hello
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ myvar=Hello
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ myvar=HeLLo
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ declare -u MYVAR
gphani@icme:~$ MYVAR=HELLO
gphani@icme:~$ echo $MYVAR
HELLO
gphani@icme:~$ MYVAR=hello
gphani@icme:~$ echo $MYVAR
HELLO
gphani@icme:~$ MYVAR="hello, world"
gphani@icme:~$ echo $MYVAR
HELLO, WORLD
gphani@icme:~$
```



```
gphani@icme:~$ myvar=hello
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ myvar=Hello
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ myvar=HeLLo
gphani@icme:~$ echo $myvar
hello
gphani@icme:~$ declare -u MYVAR
gphani@icme:~$ MYVAR=HELLO
gphani@icme:~$ echo $MYVAR
HELLO
gphani@icme:~$ MYVAR=hello
gphani@icme:~$ echo $MYVAR
HELLO
gphani@icme:~$ MYVAR="hello, world"
gphani@icme:~$ echo $MYVAR
HELLO, WORLD
gphani@icme:~$ declare +u MYVAR
gphani@icme:~$ MYVAR="hello, world"
gphani@icme:~$ echo $MYVAR
hello, world
gphani@icme:~$ declare +l myvar
gphani@icme:~$ myvar=HELLO
gphani@icme:~$ echo $myvar
HELLO
gphani@icme:~$
```

So, let us go and explore that declare and then I would like to insist that numbers only should be stored in a variable called mynum, so mynum should only have integers. So, mynum is equal to 10 echo mynum and you will see that the variable is storing it correctly. Now, I try to store a string in that and now what happens is of course, it does not comply upfront, but you see that it has stored 0. So, you can see that when you try to store a string in a variable which is designated only to store integers then things will go wrong and you normally get 0 if you try to assign it.

So, now, let us do the same thing with the other variables with other restrictions, so the minus i restriction is for integers. So, let us say minus l for lowercase characters, so I will say myvar, so myvar is equal to hello echo myvar and you see that it is able to reproduce quite well. Now, I would like to store something else in that.

So, capital HELLO and you see that the capital H has been translated to small H and the string has not really changed its character. So, the same string is available, so it is not as problematic as in integers where we actually lost that particular value and we got a 0 as the value of the variable.

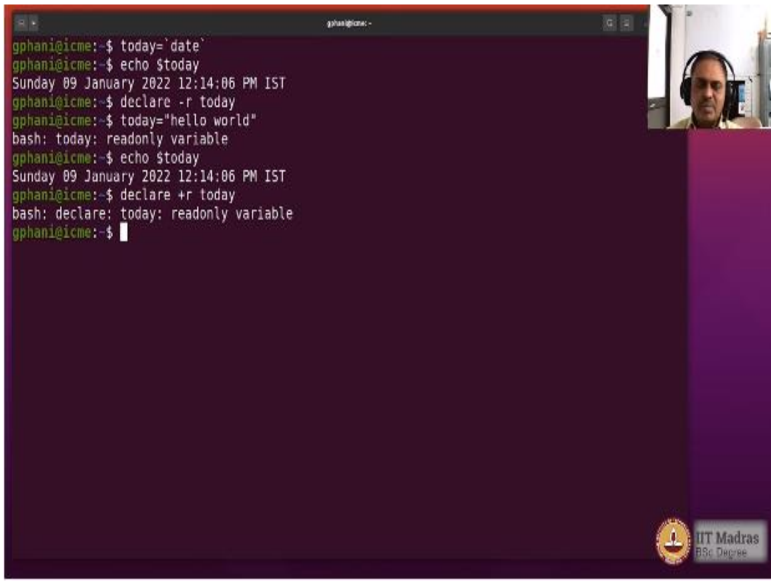
So, here that is not the problem only the case has been adjusted to fit to what is the declared case for the variable and that is quite fine. So, we could actually also see that you could mix the cases, anywhere the string and that would still ensure that the string is only having lowercase characters, this is very convenient where you are expecting for example, all the characters to be lower cases there is no point in trying to search for uppercase and then replace them etcetera, you just simply put the restriction and then assign the values and the values will be adjusted as per the declared case of that particular string variable.

Now, let us declare the string variable with the uppercase characters, so for that I would actually say myvar is my uppercase variable. So, myvar is equal to and here I would type HELLO echo and it will contain the uppercase characters as we have originally assigned. Now, if I happen to assign a small case, lower case and they would automatically get converted to uppercase. So, again, try that out you see that the entire string has been converted to uppercase because we have made the restriction here to be uppercase characters. So, these restrictions are quite useful in doing some translations of the characters automatically while the value has been assigned.

So, we could actually also switch off that particular attribute. So, I can say declare plus u so I am just switching off the uppercase restriction for myvar and then I would assign the string which is having lowercase characters. So, now I have put a plus u which means I have removed the restriction. So, if I echo then I would get the same string with lower case characters so the restriction has now been removed.

So, the same thing is possible with the other variables that we have done so declare plus l myvar and myvar is equal to, so it is now accepting uppercase characters because the restriction of lowercase has been removed with a plus l. it is a little different because plus l normally means as if you are adding some attributes, but in this case it is actually removing the attribute of forcing the lower case characters for that particular variable.

(Refer Slide Time: 34:46)

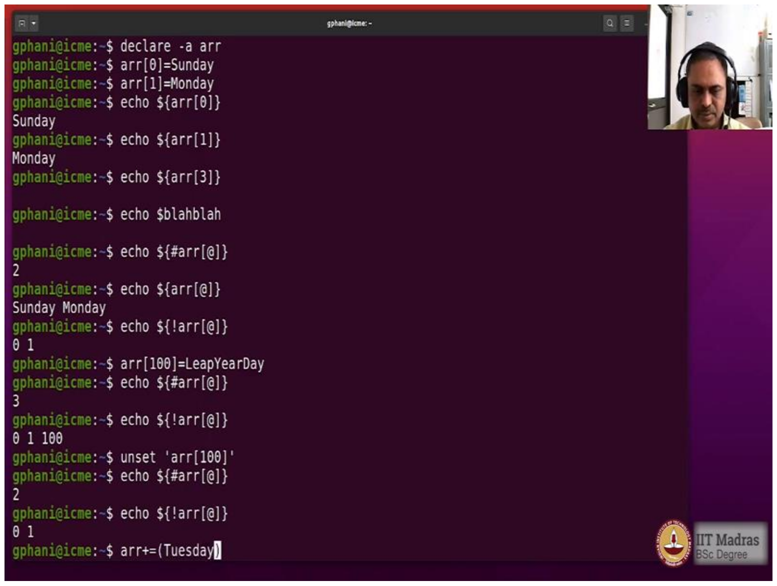


```
gphani@icme:~$ today=`date`
gphani@icme:~$ echo $today
Sunday 09 January 2022 12:14:06 PM IST
gphani@icme:~$ declare -r today
gphani@icme:~$ today="hello world"
bash: today: readonly variable
gphani@icme:~$ echo $today
Sunday 09 January 2022 12:14:06 PM IST
gphani@icme:~$ declare +r today
bash: declare: today: readonly variable
gphani@icme:~$
```

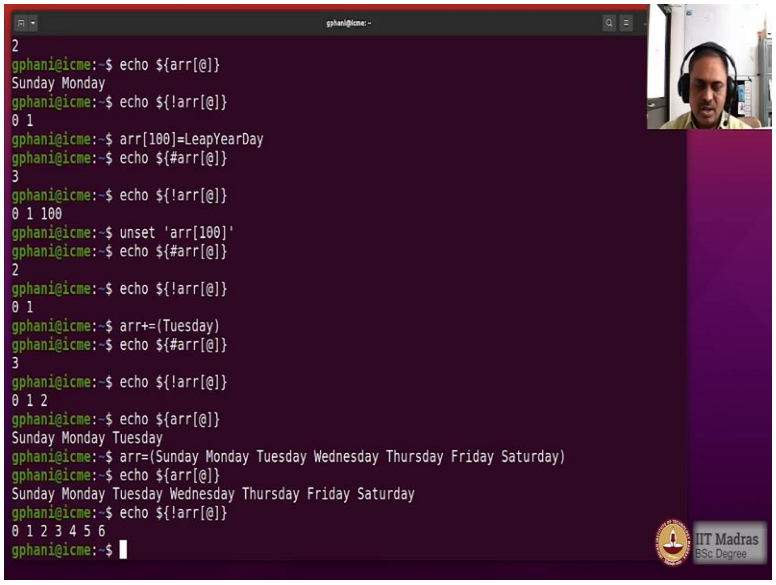
Now, there is another attribute which is also to ensure that you do not change the value of that particular variable once it is made available to the user, that is basically the read only attribute. So, let us say today is equal to date, so echo dollar today so the date is fixed. Now, I would like to make this particular variable today as a read only, so I will say declare minus r today. So, you see here I cannot now change it, so it prevents me from changing the value of the variable today because it is read only so it does not get changed.

Now, I can also not change the attribute itself, even the attribute changing also is not possible which means that once you set a particular variable as read only, then by and large it is safe because you cannot change the attribute, you cannot also change the value. So, it is a very nice way by which if you have a script that is passed on to a user and you want to have some variables in the script to be available to the user but only as a read only feature then you can have this attribute and then pass on and then those variables cannot be modified by the user either intentionally or by mistake.

(Refer Slide Time: 36:38)



```
gphani@icme:~$ declare -a arr
gphani@icme:~$ arr[0]=Sunday
gphani@icme:~$ arr[1]=Monday
gphani@icme:~$ echo ${arr[0]}
Sunday
gphani@icme:~$ echo ${arr[1]}
Monday
gphani@icme:~$ echo ${arr[3]}

gphani@icme:~$ echo $blahblah

gphani@icme:~$ echo ${#arr[@]}
2
gphani@icme:~$ echo ${arr[@]}
Sunday Monday
gphani@icme:~$ echo ${!arr[@]}
0 1
gphani@icme:~$ arr[100]=LeapYearDay
gphani@icme:~$ echo ${#arr[@]}
3
gphani@icme:~$ echo ${!arr[@]}
0 1 100
gphani@icme:~$ unset 'arr[100]'
gphani@icme:~$ echo ${#arr[@]}
2
gphani@icme:~$ echo ${!arr[@]}
0 1
gphani@icme:~$ arr+=(Tuesday)
```



```
2
gphani@icme:~$ echo ${arr[@]}
Sunday Monday
gphani@icme:~$ echo ${!arr[@]}
0 1
gphani@icme:~$ arr[100]=LeapYearDay
gphani@icme:~$ echo ${#arr[@]}
3
gphani@icme:~$ echo ${!arr[@]}
0 1 100
gphani@icme:~$ unset 'arr[100]'
gphani@icme:~$ echo ${#arr[@]}
2
gphani@icme:~$ echo ${!arr[@]}
0 1
gphani@icme:~$ arr+=(Tuesday)
gphani@icme:~$ echo ${#arr[@]}
3
gphani@icme:~$ echo ${!arr[@]}
0 1 2
gphani@icme:~$ echo ${arr[@]}
Sunday Monday Tuesday
gphani@icme:~$ arr=(Sunday Monday Tuesday Wednesday Thursday Friday Saturday)
gphani@icme:~$ echo ${arr[@]}
Sunday Monday Tuesday Wednesday Thursday Friday Saturday
gphani@icme:~$ echo ${!arr[@]}
0 1 2 3 4 5 6
gphani@icme:~$ 
```

Now, let us explore the concept of arrays. There are two types of arrays, index arrays as well as the associate arrays, but is basically the regular arrays with an integer as the index or a string as an index in the case of the hashes associate arrays. So, this is done by using the declare option, so we will say declare minus a and then array so arr is my array.

So, now arr happens to be an array and therefore we can actually start assigning values to it so arr and then the array index can be given in the square brackets, so we would give the first index as 0 for example. And the equal to sign has to be given just like any other shell variable, so no space between the variable name and the equal to sign. So, those restrictions are applicable. So, we will have the first value as Sunday and array 1 is Monday.

So, now I would like to check what are the variables stored in this particular array, so we could actually inspect those now, dollar array and then let us say 0, so it will give you the 0th item and the first item available, now if you happen to give any arbitrary index and if it is not there nothing dangerous will happen, it is just simply going to give you a null because such a variable does not exist and this is true for any other shell variable. So, if you ask for any shell variable which did not exist it does not crib much, it just simply give you a null on the screen and moves on.

Now, how do I know how many elements are there in the array? Hash array and here in the index position I just put a symbol at to indicate all the possible keys are to be used for that purpose and now we get a number 2, so that means there are two values that are stores in the array called arr, now what are those also can be asked, so without the hash basically the variable itself it used, so you could then ask for what are all those and you see that both the values are now displayed on the screen.

You could also ask what are the indices that are actually used. So, we already have come across the exclamation mark it is basically to inspect the names of the variables so in this case being an array the name is basically the index and therefore that will be returned and you see the indices that are being used are 0 and 1.

Now, why is that actually important? If you knew the size of the array, is it not obvious that you could just simply go up to the size of the array, so here you have 2 which means that 0, 1 should be the two indices that are possible, the answer is no because you can actually have any other index also used without actually filling up intermediate indices, so I could actually have something like this array of 100 is equal to, and I can make some random string.

Now, I can ask how many elements are there, so it says there are three elements, so what are the indices? It is not 0, 1 and 2 but it is 0, 1 and 100. You can see that the indices are 0, 1 and 100, so it very important for us to inspect what are the indices because those indices are not necessarily continuous. So, this is one of fundamental difference between the arrays in shell versus arrays in C language or FORTRAN language etcetera where the indices are all continuous from 0 to the size of the array.

Now, let us say you would like to delete a particular element from the array, so in this case I would like to delete the 100th element, so you could do it with unset, so unset and then array 100, so I pass on along with the index so that the entire thing is interpreted as if it is a variable name and that is removed, so now I can ask how many elements are there, there are two elements and what are those indices, and 0 and 1. So, the 100th elements has been removed by using the unset command and the index which we would like to remove is indicated with in the square brackets as if the entire thing is a variable name.

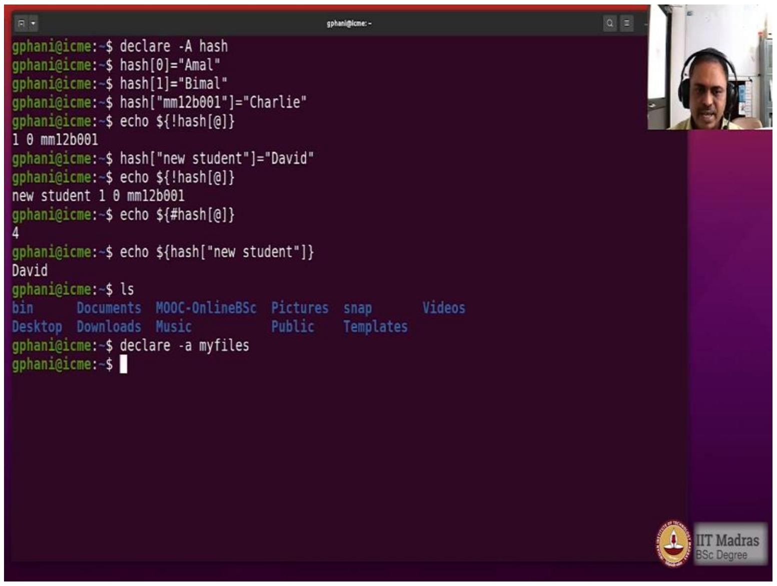
Now, can I append something to the array that is used in the same syntax as in C language, so you could have array plus is equal to and values of the array are generally given in parenthesis so you could have something like this, now you could then ask how many elements are there in the array, there are three elements and what are those indices, 0 and 1 and 2 and what are those variables, so you could say Sunday, Monday, Tuesday.

So, you could see that you can append a value to array, you can insert a particular value in the array with any arbitrary index, you could also remove it from the particular array, you could inspect what are all the indices that are being used and you can also inspect what are the values that are used in an array, so you can see that the array concept in the shell environment is quite rich.

So, you can also populate an array in one go, so you could do that in this manner for example, so array is equal to and in brackets you could actually give the values, and now you could ask what are those variables, so you have got all those variables and what are all the indices, so indices go in the sequence. So, if you have populated the array in one go then the indices are sequential because that is the most obvious thing to do in such a situation but there is no restriction for you to stick to it, you can actually insert anything at any position.

There are also some situation where you would like to have the index of an array not as a integer but as a string, so like for example, a roll number which is string can be then mapped with the name of the person for example. So, you could also declare them as associate arrays or associative arrays or hashes.

(Refer Slide Time: 43:53)



```
gphani@icme:~$ declare -A hash
gphani@icme:~$ hash[0]="Amal"
gphani@icme:~$ hash[1]="Bimal"
gphani@icme:~$ hash["mm12b001"]="Charlie"
gphani@icme:~$ echo ${!hash[@]}
1 0 mm12b001
gphani@icme:~$ hash["new student"]="David"
gphani@icme:~$ echo ${!hash[@]}
new student 1 0 mm12b001
gphani@icme:~$ echo ${#hash[@]}
4
gphani@icme:~$ echo ${hash["new student"]}
David
gphani@icme:~$ ls
bin      Documents  MOOC-OnlineBSc  Pictures  snap       Videos
Desktop  Downloads  Music           Public    Templates
gphani@icme:~$ declare -a myfiles
gphani@icme:~$ 
```

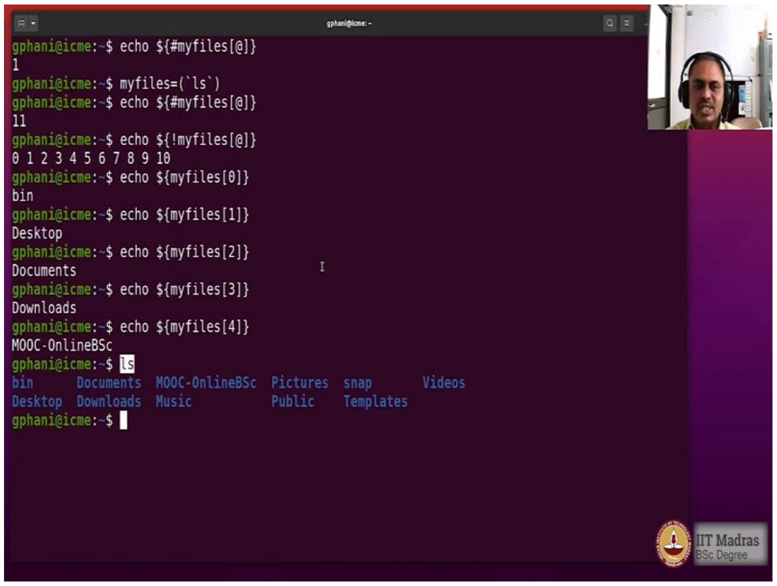
So, let us have that as a hash, declare minus a hash, so hash is now a name of variable and it will contain sequence of values which can be indexed using strings and not necessarily using only integers, integers are also possible. So, let us say 0 is equal to and the first value I would let us give a name Amal hash 1 is equal to Bimal.

Now, comes the idea, this is like an array but it’s a associative array so you can use indices which are not necessarily integers but also strings, so I will use a string which looks like a roll number, so that is like a roll number typically and now you can see that it is accepting that and now look at the keys that are used, so this bang or exclamation mark tells you to inspect the name of the arrays and in this case it say array and therefore the indices are then extracted, the at symbols tells you to run through the entire array, so now you can see that the three indices that are used are 1, 0 and mm12b001, so you could actually mix indices to be just strings or numbers.

So, I would actually have the strings to be anything else new student, and you could now see that there are four indices of course, the space may appear as if it is 5, but you can actually count, so let us do the counting here, so you see there are only four of them, and each of them we can then go on to inspect the value and here you should give the key, so you could give the key as let us say and it will give you the value. So, you can see that they keys need not be simple strings, they can be complicated strings with spaces in between, it could also be numbers and you are now free to store a hash of set of values that you would like to keep for your work.

So, you could actually have this as a additional feature to write scripts which can be used to manipulate the output of a command the way you would like to have for your scripting usage. So, when I type ls I have got the list of the files there, declare minus a myfile.

(Refer Slide Time: 47:11)



```
gphani@icme:~$ echo ${#myfiles[@]}
1
gphani@icme:~$ myfiles=(`ls`)
gphani@icme:~$ echo ${#myfiles[@]}
11
gphani@icme:~$ echo ${!myfiles[@]}
0 1 2 3 4 5 6 7 8 9 10
gphani@icme:~$ echo ${myfiles[0]}
bin
gphani@icme:~$ echo ${myfiles[1]}
Desktop
gphani@icme:~$ echo ${myfiles[2]}
Documents
gphani@icme:~$ echo ${myfiles[3]}
Downloads
gphani@icme:~$ echo ${myfiles[4]}
MOOC-OnlineBSc
gphani@icme:~$ ls
bin      Documents  MOOC-OnlineBSc  Pictures  snap       Videos
Desktop  Downloads  Music           Public    Templates
gphani@icme:~$ 
```

So, here you have got 11 and you could inspect what are all those indices, so it goes from 0 to 10 and what are those you could also inspect them, so bin and then desktop, then documents and downloads and so on.

So, you could see that there are 11 folders in the directory that I have with the ls command, I am getting 11 entries, and those 11 entries are now indexed from 0 to 10 in the array called myfiles and I can inspect each of them one after another in this manner, so if I have a script that runs through each of this directory to perform some action, it is now quite easy because I have them in array and I can run to those arrays.

Now, to run through a array using a index you need to have a concept of loop and that brings us to the shells scripts in a text mode, so we will learn that in the upcoming classes where we will be able to write a set of commands including the looping and conditional statements etcetera, so we will explore that as we go along but these basic ideas about the variables and their values will be quite use for us to imagine how we can accomplish the task at hand.