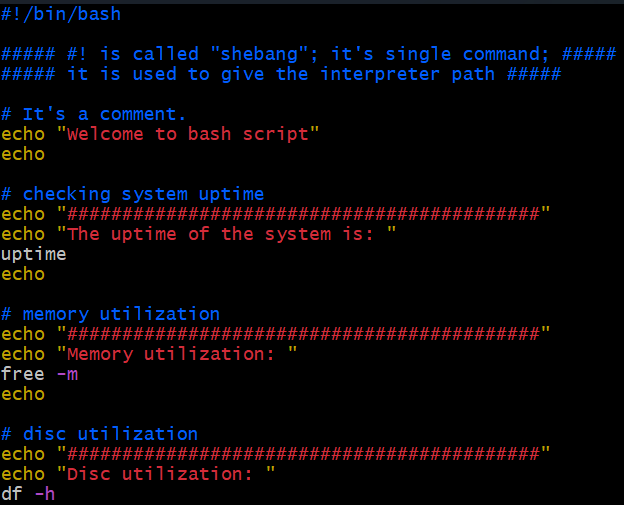
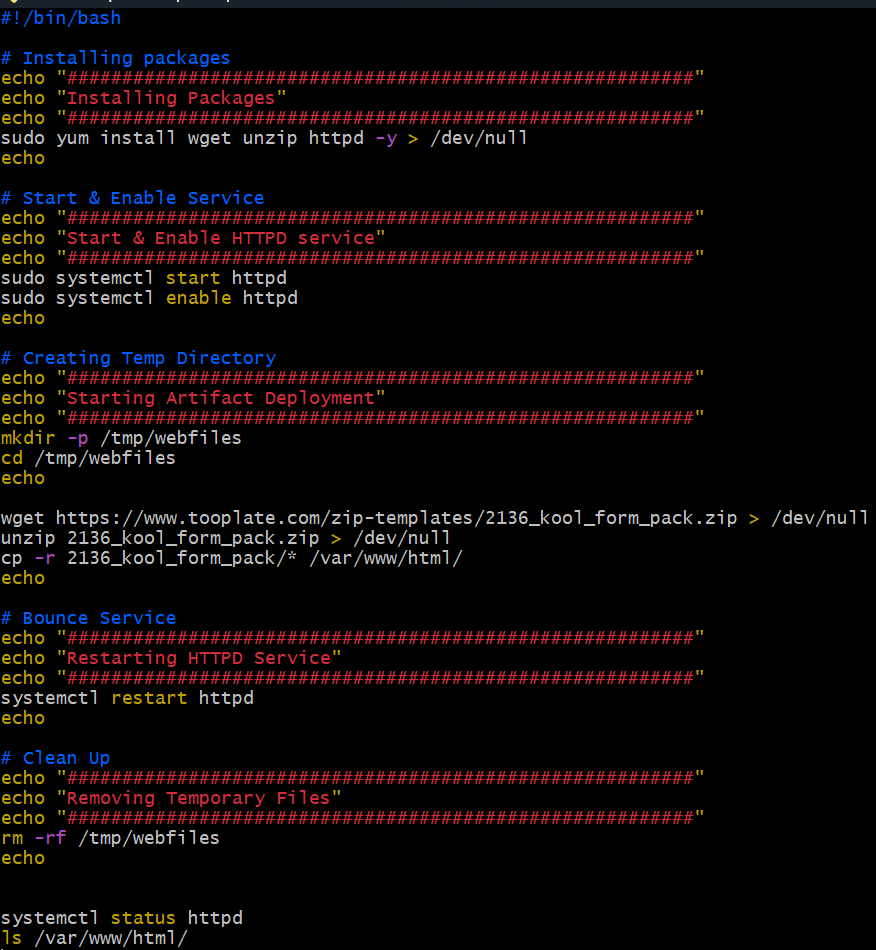
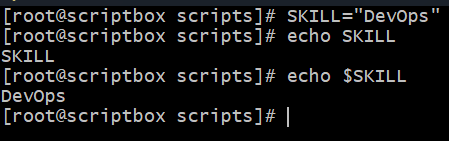
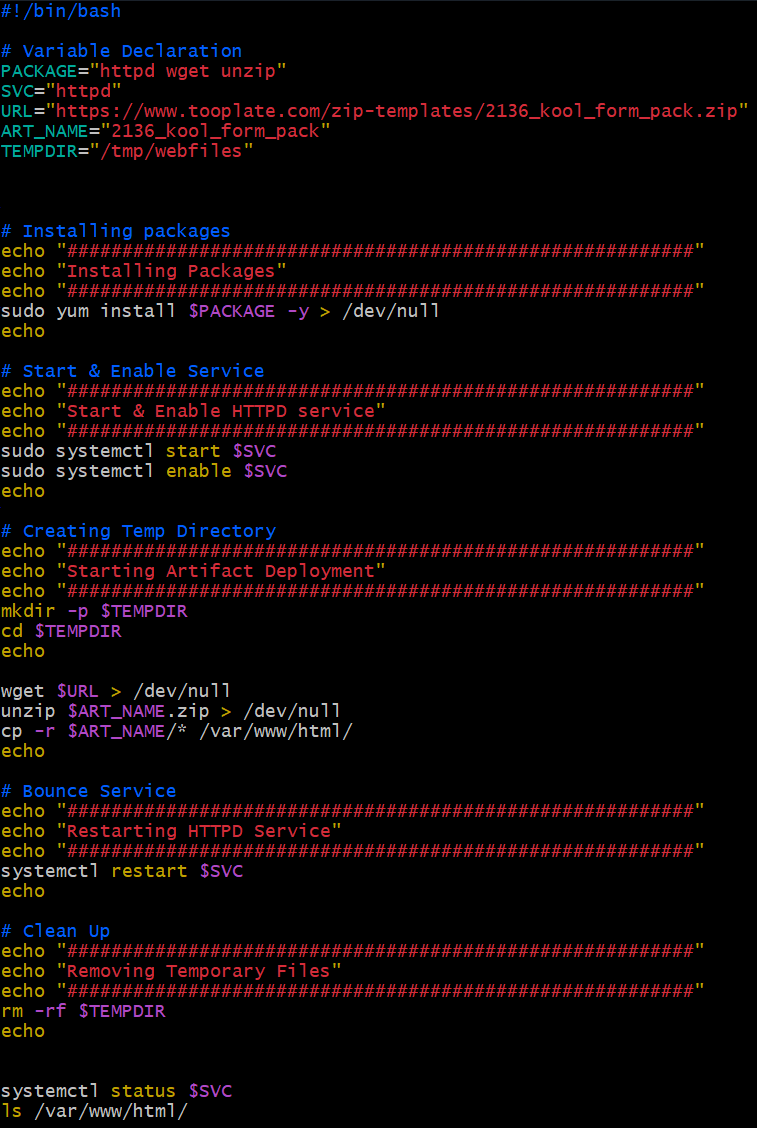
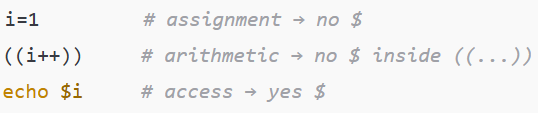
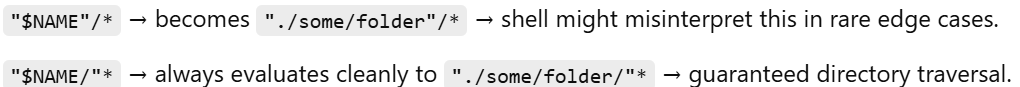
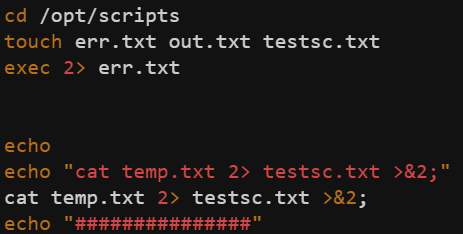
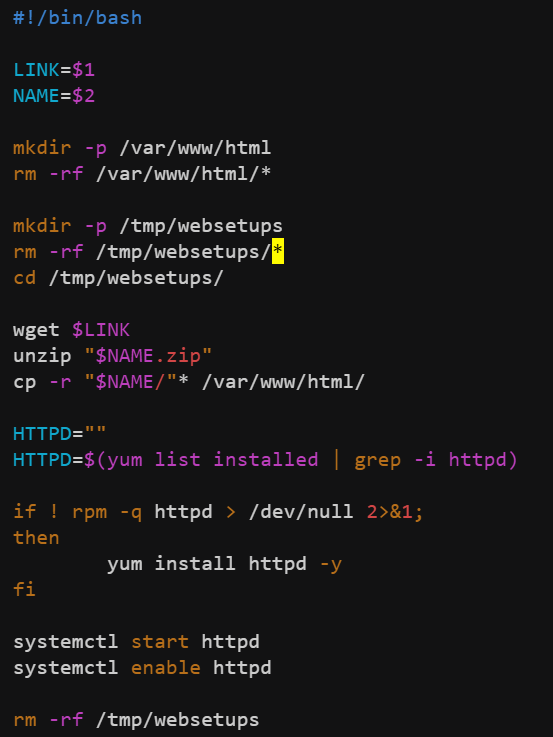
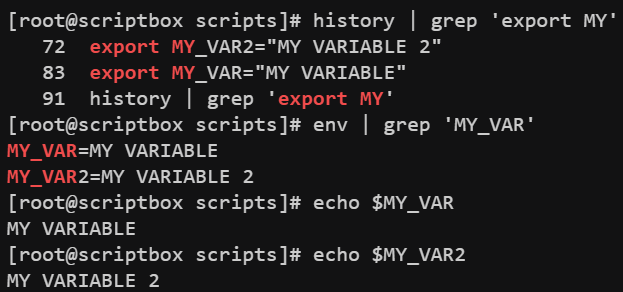
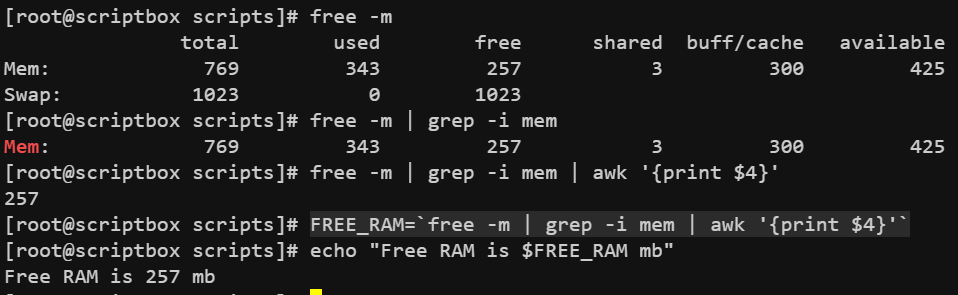
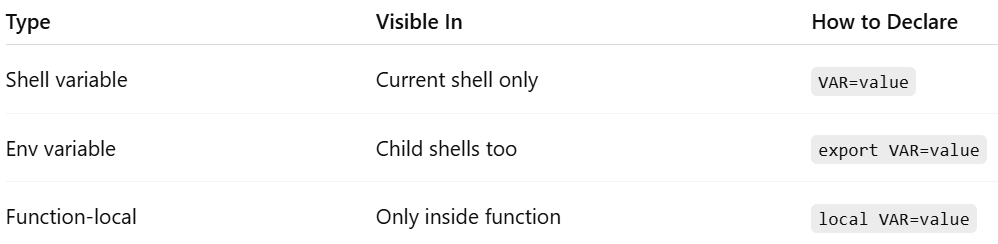
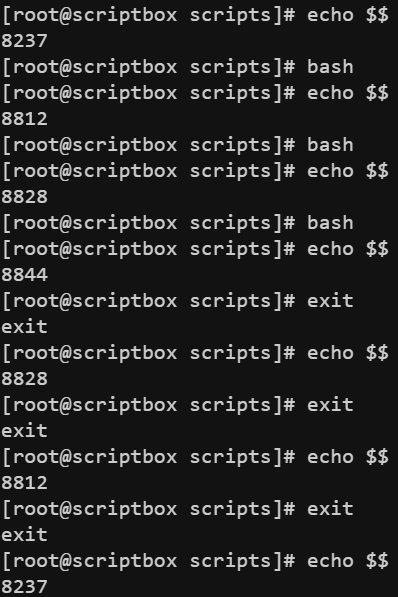
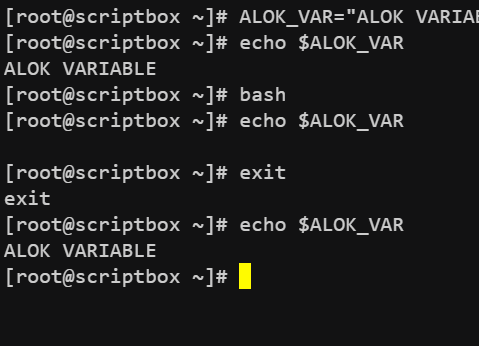
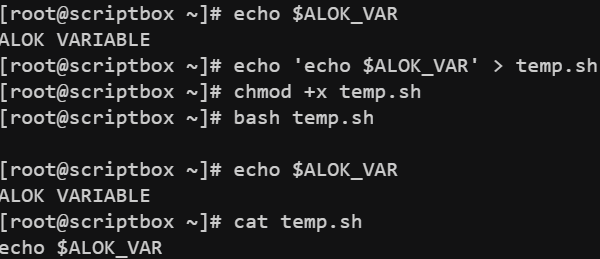
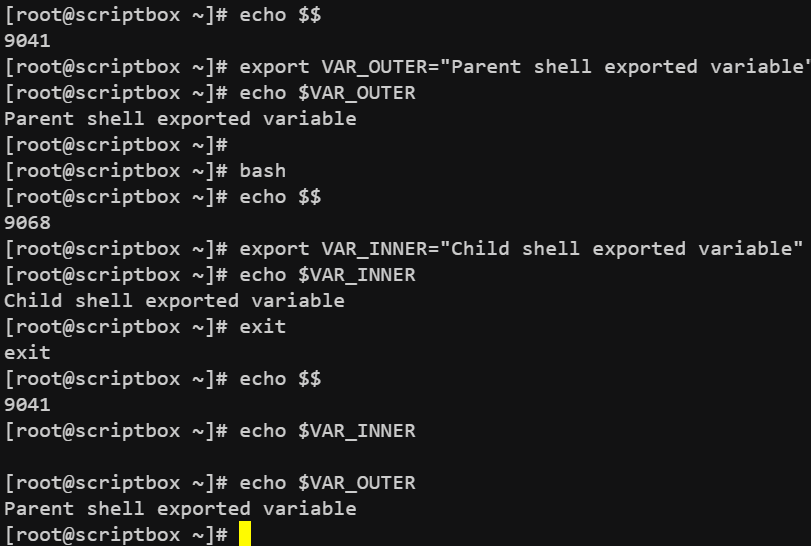
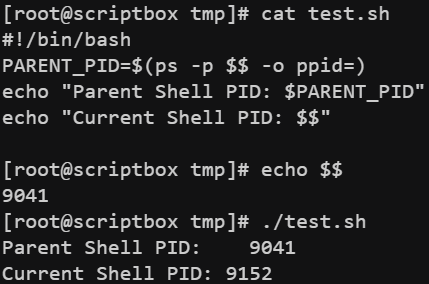
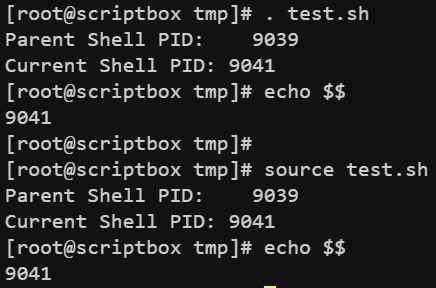
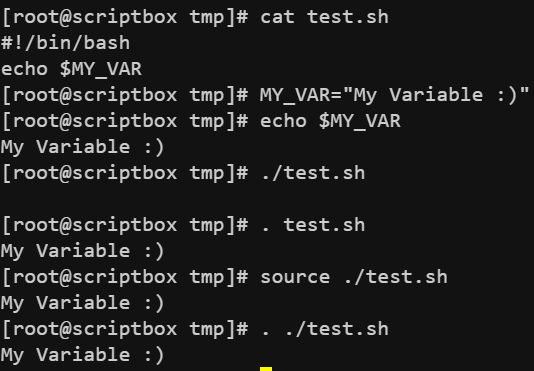
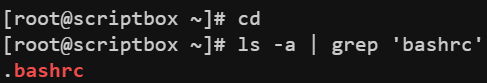
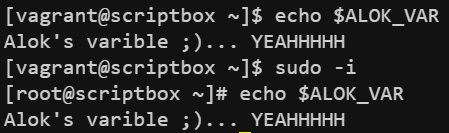
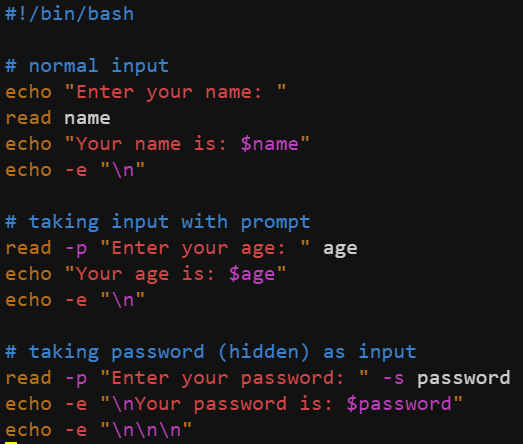
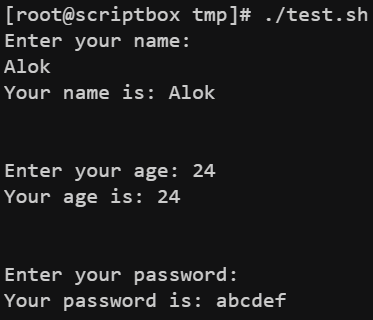
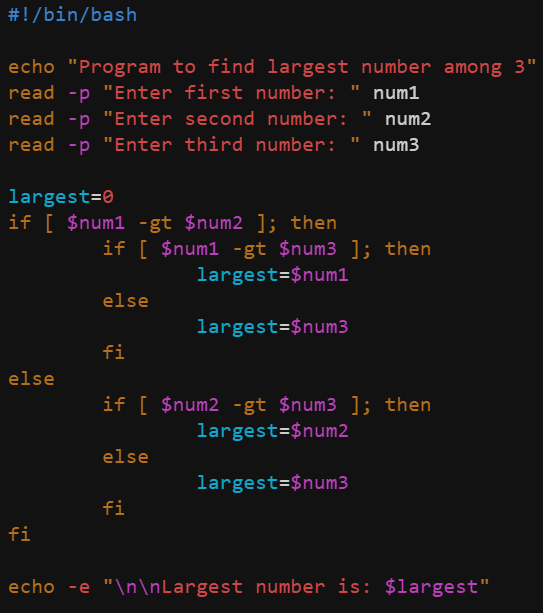
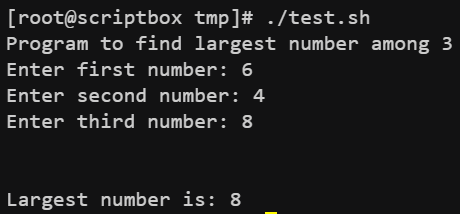
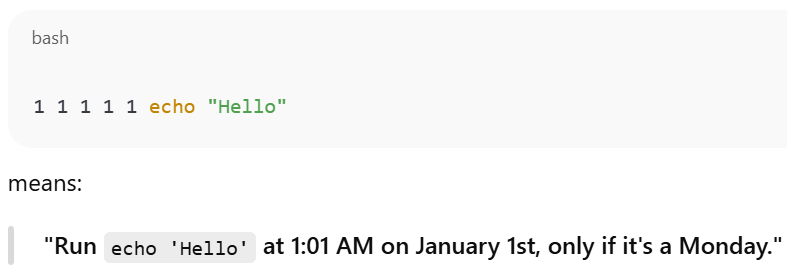
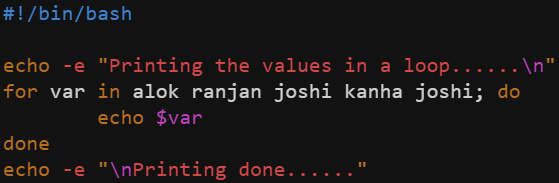
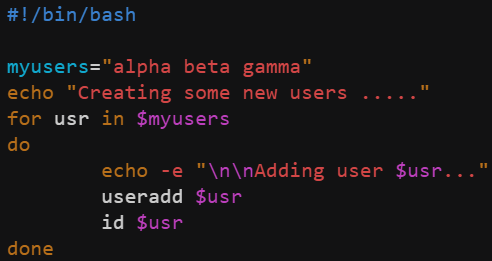
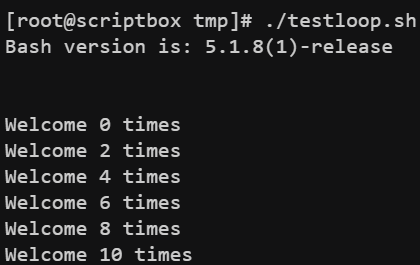
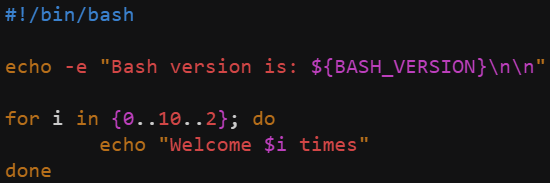
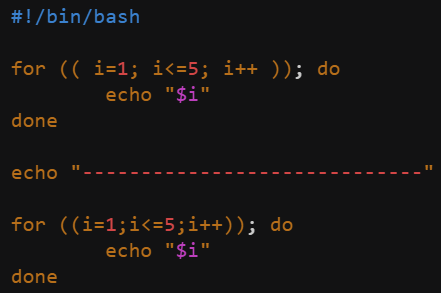
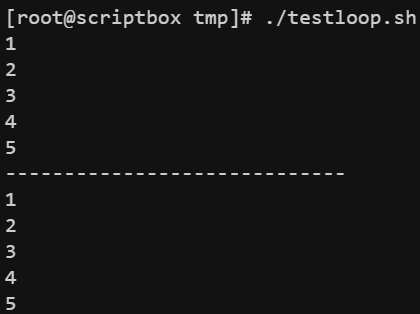
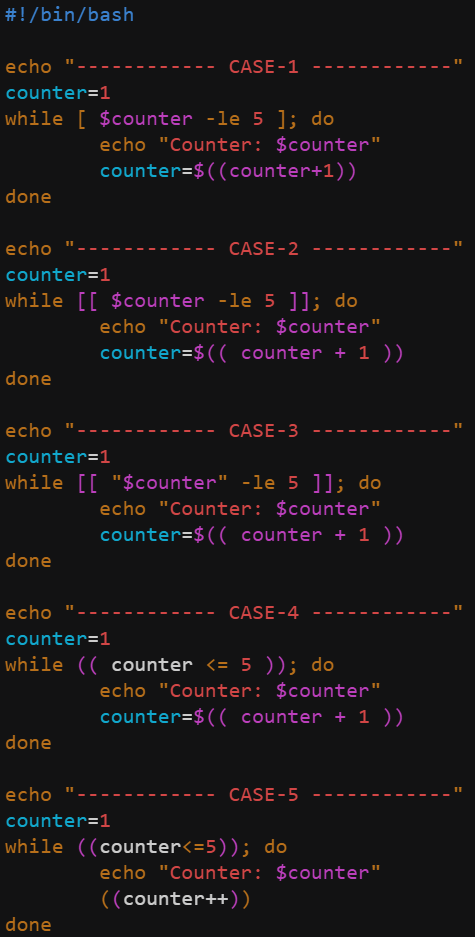
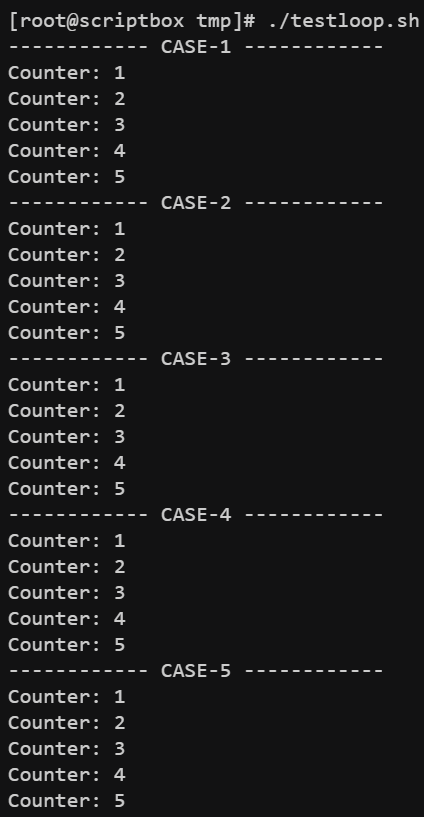
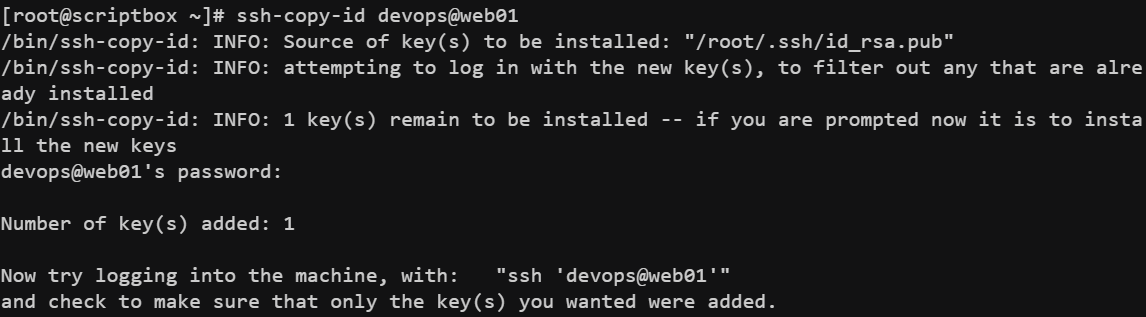
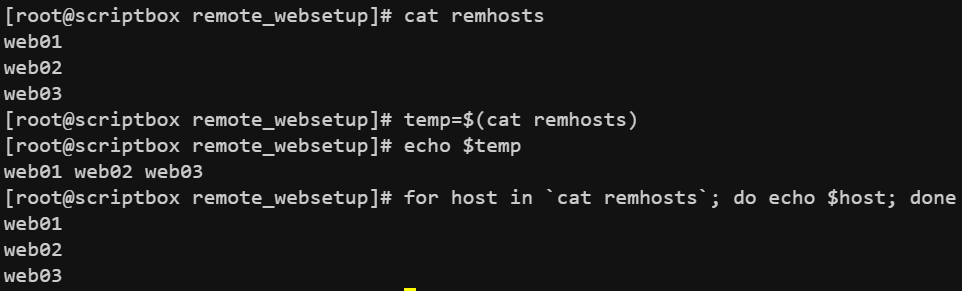
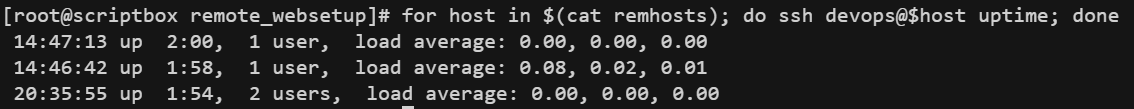
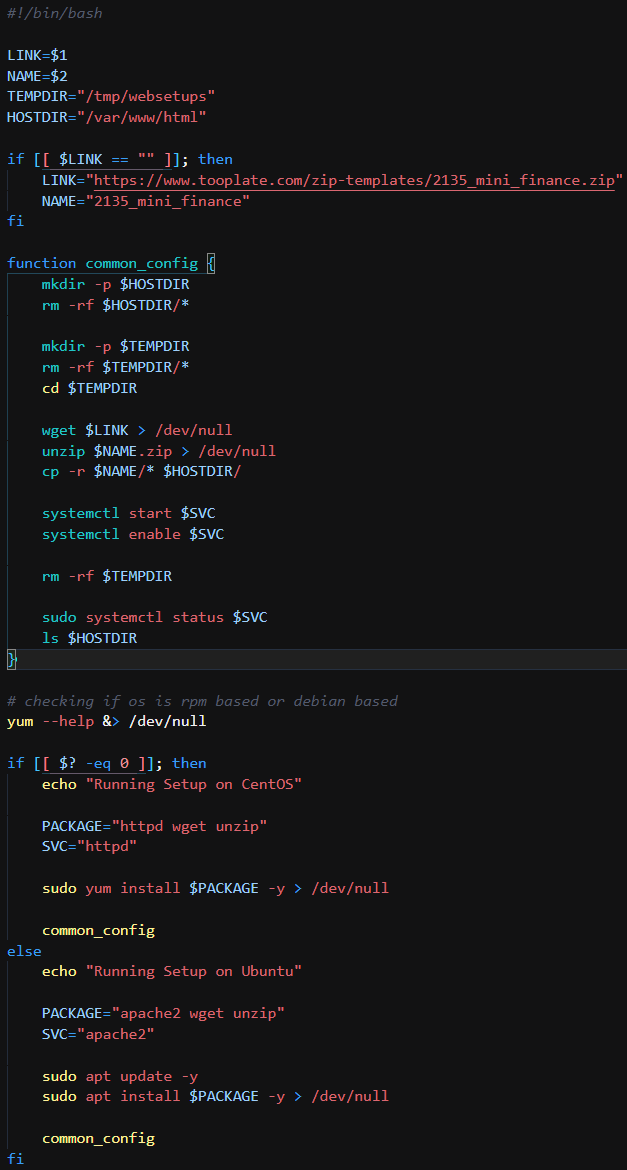
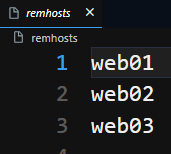
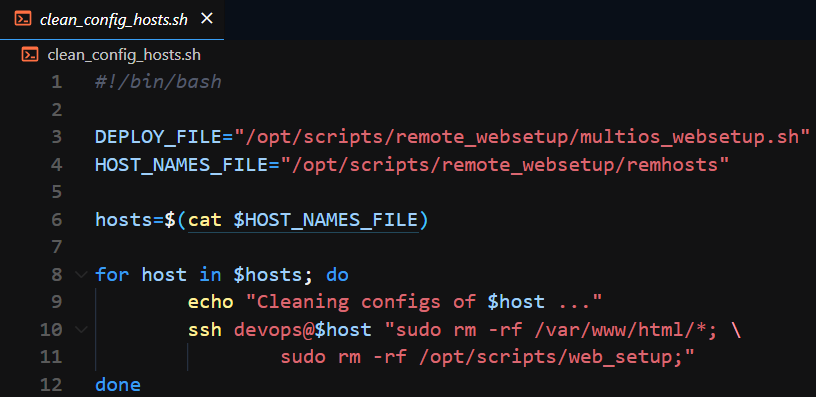
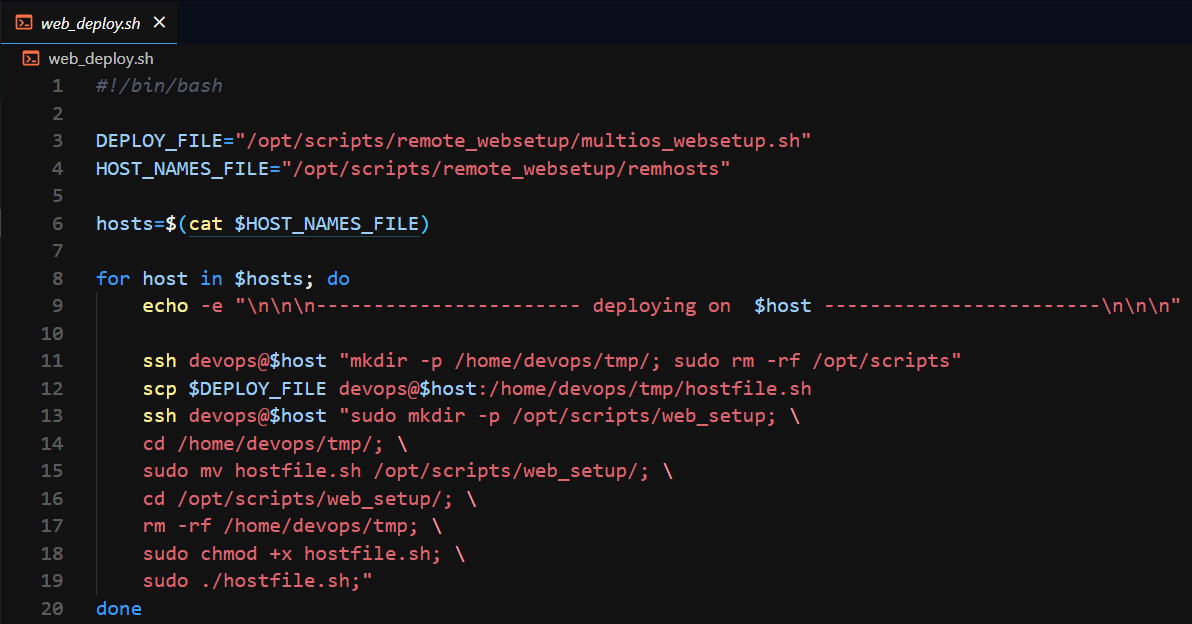
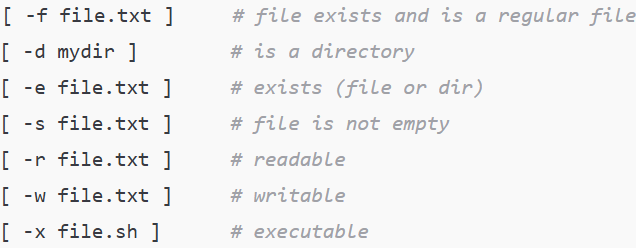
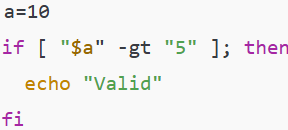
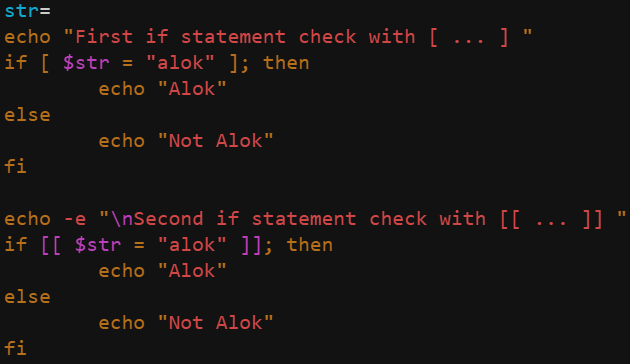
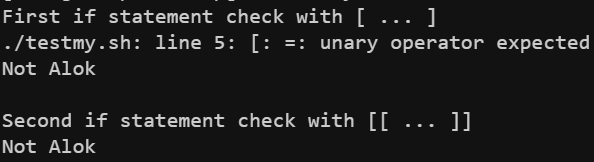
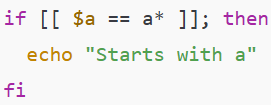
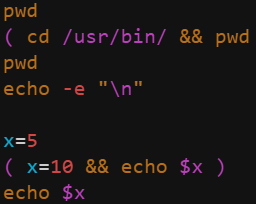
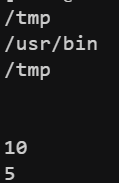
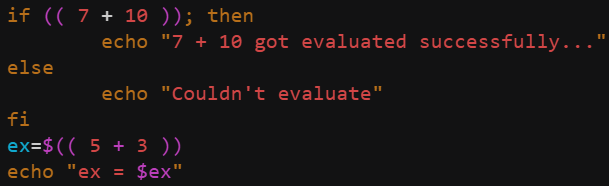
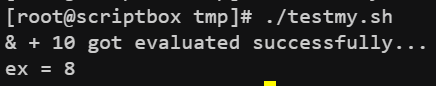
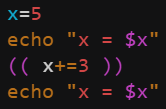
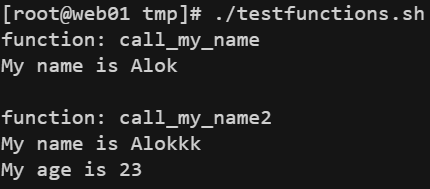
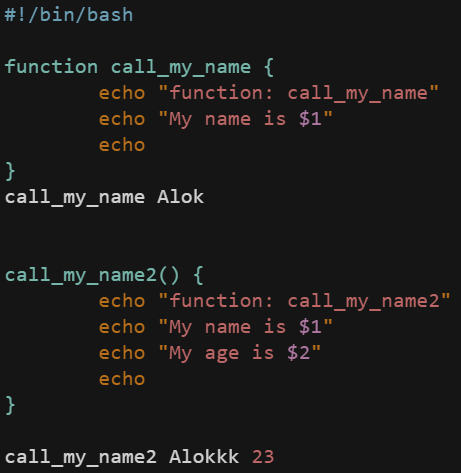
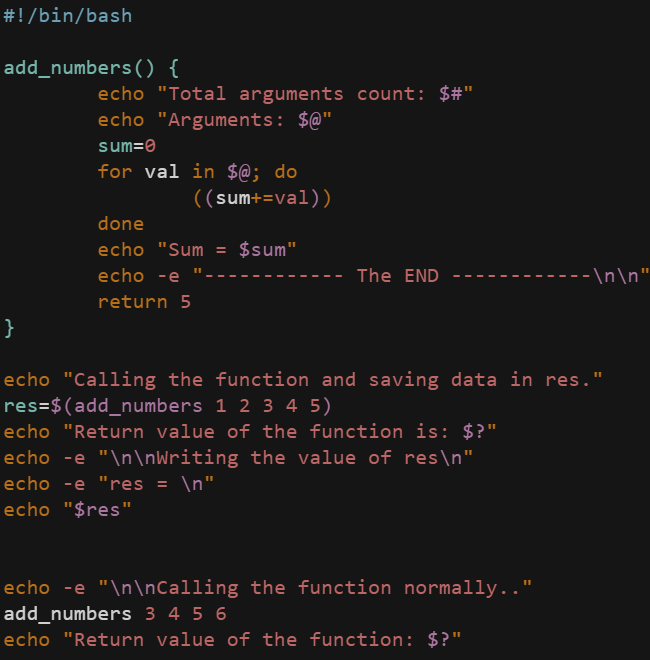
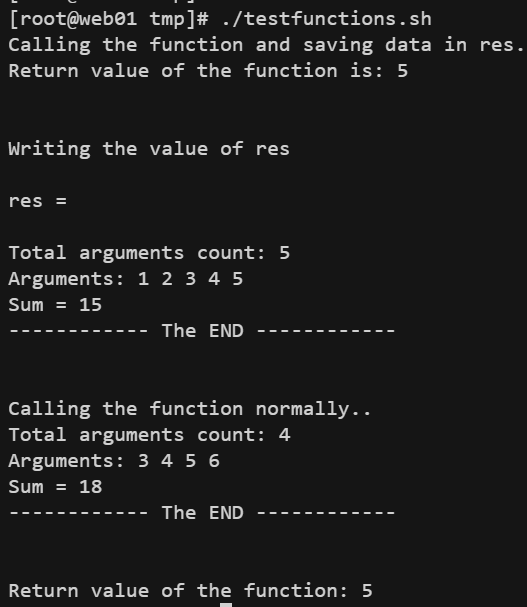
* ***#!***  is called shebang. It tells the system which ***interpreter*** will be used to run the script.
  + 
* 
* Variables:
  +  (without ***$*** it’s just a normal string, not a variable)
* Using variables:
  + 
* ✅ Short Rule:
  + Use ***$i*** when you're accessing the value of the variable.
  + Use just i when you're declaring, assigning, or iterating over it.
  + Use ***$*** whenever you would expect to "see the value" — like echo, math comparisons, etc.
  + No ***$*** when you’re giving it a value or defining the loop variable.
  + 
* 
  + Don't quote the **\*** if you want shell expansion.
  + Quote the variable part only, like this:**"$NAME/"\***
  + 
  + Only bcs of ***\**** we are not able to write ***“$NAME/\*”*** .
  + If dir1=abc, dir2=def; we can write ***cd “$dir1/$dir2”*** , it’ll be correct. But when we write ***\**** inside a sting, it’ll not get expanded.
* ***Advanced redirection:***
  + 
  + Let’s first understand what is ***>&2*** 
    - ***command 2> test.txt >&2***
    - Here **2>** means **stderr** redirection
    - **>&2**
      * > means stdout redirection
      * ***&2*** means the path where ***2>*** is referring (here **test.txt**)
      * ***>&2*** means, stdout will go to the path where ***>2*** is going i.e. **test.txt**
    - Similarly ***command > test.txt 2>&1***
      * It’s also same;
      * ***2>&1***
        + Means ***stderr*** will to to the path where ***stdout*** is going i.e. test.txt
    - *So simply* ***&x*** *means the path which is getting referred by* ***x>*** *(****x=1, 2****)*
  + So, now let’s see the above screenshot.
    - I have set all the future ***stderr*** path to ***err.txt***
    - Then I wrote ***cat temp.txt 2> testsc.txt >&2***
    - Here, ***2> testsc.txt*** will override the ***err.txt*** for this command, so for this command, ***stderr*** is referring to ***testsc.txt***.
    - So, ***>&2*** means ***stdout*** will go t o the path where ***stderr*** is going i.e. ***testsc.txt*** (for this command only.. otherwise ***err.txt***)
* In bash ***0: true, non-zero: false***
* if ! rpm -q httpd > /dev/null 2>&1 means if httpd is installed, it’s exit code will be ***0*** hence true.
* (script to host any site in httpd)
  + 
  + ***$1 $2*** represents the command line arguments . $0 represents the command itself.
* Variables
  + 
    - It’ll install these packages wget, httpd, unzip.
  + While declaring variables I.e. VAR1=”value1”
  + While using variables I.e. echo $VAR1 (use ***$*** while using variables)
* 
  + ***export*** command is used to set any ***env*** variable.
  + You can check all the ***env*** variables using the commsnd ***“env”***.
* **Following are the built-in shell(bash) special/system variables:**

|  |  |
| --- | --- |
| **Variable** | **Description** |
| $RANDOM | Returns a random integer between 0 and 32767 |
| $UID | User ID of the current user |
| $EUID | Effective UID |
| $HOME | Current user’s home directory |
| $PATH | Colon-separated list of directories to search for executables |
| $PWD | Present working directory |
| $OLDPWD | Previous working directory (cd -) |
| $SHELL | Path to the current shell |
| $USER | Username of the current user |
| $HOSTNAME | Hostname of the system |
| $SECONDS | Number of seconds since the shell was started |
| $LINENO | Current line number in the script |
| $BASH\_VERSION | Version of Bash |
| $BASH\_SOURCE | Filename of the current script |
| **Variable** | **Description** |
| $0 | Script name |
| $1…$9 | First to ninth argument to script |
| $# | Number of arguments |
| $@ | All arguments as separate quoted strings |
| $\* | All arguments as one word |
| $? | Exit status of last command |
| $$ | PID of the current shell |
| $! | PID of last background command |

* Command substitution:
  + Stores the output of a command in a variable
  + Use back-tick ***``*** or ***$()***
  + 
  + **NOTE: these are called command substitution. Means the output of the command will not go to the screen now, it’ll be stored in the variable only.**
* 
* Child shells:
  + 
  + ***$$*** is used to print PID of current shell. 8237(parent shell) -> 8812 -> 8828 -> 8844
  + So 3 levels of hierarchy got established.
  + You can get out of ***child*** shell to ***parent*** shell using ***exit*** command.
  + 
  + In shell script, child shell can’t excess the variables declared in parent shell.
  + 
    - By default, shell script files run in child shell. So that also can’t access the variables.
  + 
    - ***When in a shell, a variable is exported, it’ll be available to all of it’s child shells but not to parent shell. & If u export a variable and logout and again login, the variable will not be there.***
* When u run a script, it by default gets run *in a* child shell.
  + (Parent shell PID: 9041, script ran in 9152 which is a child shell)
  + 
    - If u run the script using ***.*** *(dot<space><filename>)*or ***source*** command, then it’ll be run in the current shell only.
  + (When we ran the script using ***source*** or ***.*** it accessed the variable MY\_VAR)
* In home directory of every user(root or any other user) there is an ***.bashrc*** file which is loaded (executed) after log in with that user’s shell. If u want to make a variable be accessed for that user even after logging out and logging in, you can ***export*** that variable inside that file.
  + 
  + NOTE: That variable will only be accessible by the particular user, whose **.bashrc** file had been updated.
  + If u want to make the variable accessible for all the users, then ***export*** the variable inside the file ***/etc/profile***
  +  (like this)
  + (for all user it is accessible)
  + **NOTE :** First ***/etc/profile*** file is sourced and then ***.bashrc*** file. So, if same variable is declared in both, then ***.bashrc*** will override that ***/etc/profile*** .
* Taking input from CLI:
  + (-p for prompt, -s for hidden input)
    - 
* Decision making(if, elif, else)
  + 
  + 
  + NOTE: there must be a space after ***[*** and before ***]*** in if or **elif** statements. Otherwise it’ll take ***[<any char>*** as one single command.
* **Crontab:**
  + Used to do any repetitive task.
  + ***crontab*** lets you schedule commands or scripts to run automatically at specified times and dates. It uses a background service called ***cron***.
  + 
  + Example:
    - 
* *Once per year on Jan 1st that too if it’s Monday. (which is very rare)*
* Loops:
  + The semicolon **;** before do is optional if the do is on a***new line***, but ***required*** if it's on the ***same line*** as the loop.
  + (looping through an array)
  + 
    - While assigning a variable, no space should be given i.e. **myusers = “alpha beta gamma”**. it’ll think that **myusers** is a command if space is given between **myusers** and **=**
    - Inside the for loop, **$** has to be used while accessing the list or array. As for accessing, **$** has to be used.
  + 
    - Here, {0..10..2} means **0 to 10, step=2**
    - Here, I wrote **do** in the same line. So, I had to put one semicolon  **;**
    - (output)
  +  
    - gave to examples to show that space is not required here
  +  
    - Look case-2 & case-3 properly.
    - You can’t write count=count+1 or count=$count+1
* Remote Command Execution:
  + From one vm (let scriptbox), you can do **ssh vagrant@web01** like this to enter to the vagrant user shell of the vm web01.
  + There if you execute **sudo -i** then it’ll switch root user shell of **web01**.
  + In case of ubuntu, remote connection is disabled by . To enable this, update the file **/etc/ssh/sshd\_config**. (PasswordAuthentication yes) inside that file.
  + Let I execute one command **ssh devops@web01 uptime**, it’ll login to **devops** user shell inside **web01** vm, execute the command **uptime**, and get back to the current user’s shell.
  + Every-time you want to **ssh** you have to enter the password. So, ssh key exchange is used, however it is more safer. s**sh-keygen** is the command to generate the ssh key.
  + Then **ssh-copy-id devops@web01** (means **ssh-copy-id <username>@<vm name>**)
    - 
  + Now when we’ll login the user (devops@web01) it’ll not ask the password for this.
  + It basically execute  **ssh -i .ssh/id\_rsa devops@web01**  when we execute  **ssh devops@web01** .
  + 
    - If there are so many hosts, you can’t ssh them all manually as it’ll consume a lot of time. Better to use a loop for those as mentioned in the above screenshot.
  + 
* **scp** command:
  + It is used to **download/upload** (basically **copy**) the files between **current** & **remote** machines.
  + It uses **ssh** protocol.
  + **scp <path from> <path to>** 
    - **scp note.txt alok@web01:home/tmp/**  (upload)
    - **scp [alok@web01:home/tmp/note.txt](mailto:alok@web01:home/tmp/note.txt) ./note.txt** (download)
* 
* 
* 
* 
  + It is used to deploy the website in all the hosts.
    - First create a **tmp** directory inside the home directory of **devops** user & delete the **/opt/scripts/** directory if present.
    - Copy the **hostsite file** of **current host** to that **tmp** directory of **remote host**.
    - Create a folder and move that **hostsite** file to that folder (/opt/scripts/web\_setup in my case), make it **executable** and **run** that file. Remove that **tmp** file that had been created earlier.
    - As we can’t ssh root directly (we can if **PermitRootLogin yes** inside **/etc/ssh/sshd\_config** but it’s not preferable), so use **sudo** to execute all root commands.
* ***Some important points in shell scripting:***
  + Use of ***$*** :
    - Where you are accessing a variable like **echo $my\_var**
    - Inside string i.e. **“/tmp/$dir\_name/”** (here dir\_name is a variable, written inside a *double quote*)
    - In arithmetic expression like **sum = $(( x + y ))**here ***$*** means return the output of the expression so that *sum* can store it.
    - *NOTE*: No use of **$** inside (( … )).
    - **$(( … ))** is not same as **$( ( … ) )**.
    - **$(( … ))** is arithmetic expansion and ***$( … )*** runs the command and returns it’s output. However **$( (x+y) )** will **fail** because **x+y** is **not a command**.
    - Read variable in condition i.e. **if [ $i -lt 5 ]** because here you are accessing the value of ***i*** and comparing with 5.
  + Use of **[ … ]**
    - ***if, elif, else*** conditions. i.e.**if [ $a -lt 5 ]** or **if [ $str = “alok” ]**
      * Note: **-lt, -le, -gt, -ge, -e1, -ne** are used for numeric variables
      * **=, !=, -z, -n** are used for strings.
        + if [ -z $str ] means if str is empty.
        + if [ -n $str ] means if str is not empty.
    - (for files)
    - We can combine ***||*** and ***&&*** outside the brackets i.e.
      * **if [ -f a.txt ] && [ -s a.txt ]**
    - **NOTE: [ $str==”alok” ]** is wrong. **==** should have spaces around it. **[ $str == “alok” ]** it is correct.
    - **[ -f file.txt && -s file.txt ]** it is **wrong**. && can’t be used inside [ .. ]
    - 
      * Here, it’ll become [ = “hello” ], so will give error Error: unary operator expected.
      * So, best practice is**[ “$str” = “hello” ]**
    - 
      * as we know ***-gt*** is for numeric values not for strings. But here ***a*** is a number and this string represents a number. So here ***-gt*** will work fine.
      * If ***a=”alok”*** and we have written **if [ “$a” -eq “alok” ]** it’ll give error as ***-eq*** is not for strings.
  + Use of **[[ … ]]**
    - Modern and safer version of **[ … ]**
    - Same as **[ … ]** for the strings, but there if the string is empty then we were not able to write if **[ $str = “alok” ]** kind of thing as it was giving error, but in case of **[[ … ]]** we can write that. It’ll not give any error.
      * 
      * 
    - One additional thing is ***regex matching*** & ***pattern matching***.
      * 
    - In case of **[ … ]** we were not able to use **&&**and **||** but in case of **[[ … ]]** we can use ***&&*** and ***||*** inside that.
      * **if [[ $x -gt 5 && $x -lt 20 ]]**
  + Use of **( … )**
    - It starts a subshell.
    - ***( cd /tmp && ls )*** is same as these 4 commands in shell script ***bash, cd /tmp, ls, exit***. **NOTE:** ***( …. )*** will **run** the commands in **child** shell but gives **output** in the **current** shell. Whereas **bash, cd.., exit** will **run** commands in **child** shell and also give **outputs** in that **child** shell. So, you can’t return those outputs to the current shell.
      * (code)
      * (output)
    - We can group commands with redirection.
      * ***( echo “Line 1”; echo “Line 2” ) > output.txt***
    - Can be used in pipelines
      * ***( cd /tmp && ls ) | grep config***
    - 
      * This ***&*** at the end means it run these grouped commands in the ***background***.
    -  (it is not same as $(( cd /tmp && ls ))
    - **( echo “One”; ( echo “Two” ) ) | grep T**
      * You might thinking here, echo “Two” will return “Two” so ultimately:
        + ***( echo “One”; “Two” ) | grep T***

But it becomes **One Two** at the and so doesn’t give error. Output will be “Two”

* + - * But if you have given ***( echo “One”; “Two” )*** then it’d have thrown error.
  + Use of ***(( … ))***
    - It is used for arithmetic operations and comparisons. It’s not for ***strings*** or ***commands***.
    - Used for **arithmetic operations**, **assigning values**, **comparing values**, **increment/decrement**, **while & for loops**.
    - It returns 0 (true) or 1 (false).
    - Don’t use **$** inside this.
    - ***(( 3 + 5 ))*** evaluates and return 0 or 1 (as exit status). if u want to store the result then ***sum=$((3 + 5))***
    - ***NOTE****: As it returns 0 or 1 as the exit status, so it can be used in side the* ***if else*** *statements as well to check so that we can get rid of those -gt, -eq etc etc things.*
      * (code)
      * (output)
    - Arithmetic operations
      * (code)
      * (output)
  + ***For* [ … ], [[ … ]] *spaces are required but for* ( … ) *and* (( … )) *space are optional.***
* Arithmetic operation
* **Use of {}**
  + To access the variable.
  + **${filename}.txt** here in this type of scenario it’s helpful and safer. As if we write **$filename.txt** then it’ll find the variable having name **filename.txt**
* Functions in Shell Scripts
  + 
    - These are the 2 types of function declaration. If you are giving the keyword **“function”** then the parenthesis **()** is not needed. If you are not giving the **“function”** keyword then parenthesis **()** is needed.
    - You can pass the arguments as the *cli parameters*. Any number of arguments can be passed.
    - **$0** doesn’t mean the function name here… it refers to the CLI command.. if I’ve run **./testfunctions.sh** then the **$0** inside all the function in that script will be **“./testfunctions.sh”**.
  + **$@** represents **all the arguments** of the function. (list of arguments)
  + **$#** represents the **number of arguments** of the function.
  + **NOTE** 
    - Whatever things you **print** inside that function using **echo** command **will be returned** from the function as data i.e. **my\_data=$(fun\_name arg1 arg2)**. after executing this if u check **$?** then you can see the **return value (i.e. exit code)** of the function.
    - If you use **return** command inside the function then it’ll return the **exit code** of the function.
    - **return** statement can only return numeric values from **0-255**.

* + In first case the prints inside the function was not executed bcs of command substitution **$(…)**.