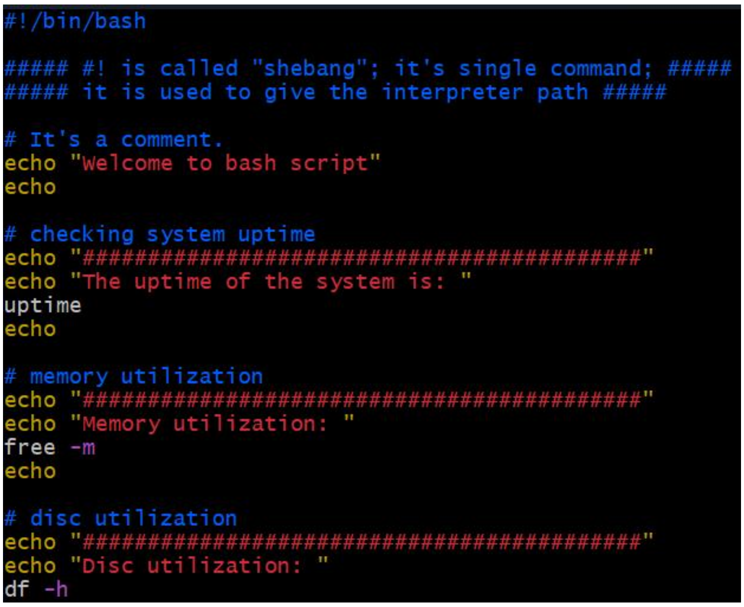
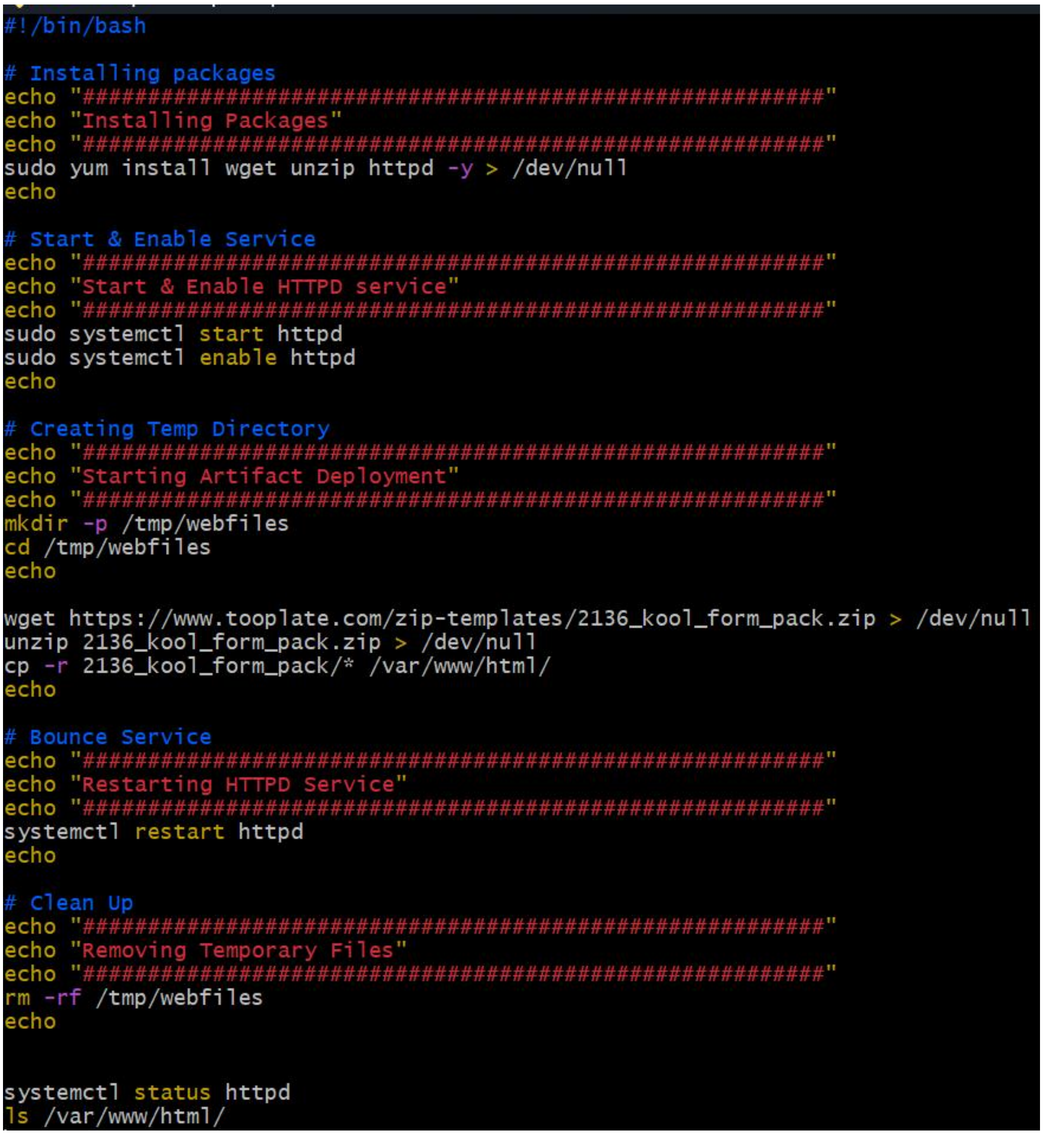
- #! is called shebang. It tells the system which ***interpreter*** will be used to run the script.

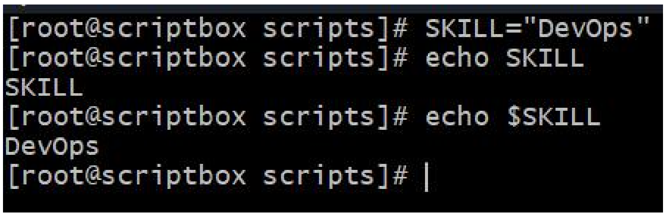


```bash
#!/bin/bash

##### #! is called "shebang"; it's single command; #####
##### it is used to give the interpreter path #####

# It's a comment.
echo "Welcome to bash script"
echo

# checking system uptime
echo "###########################################"
echo "The uptime of the system is: "
uptime
echo

# memory utilization
echo "###########################################"
echo "Memory utilization: "
free -m
echo

# disc utilization
echo "###########################################"
echo "Disc utilization: "
df -h
```



```bash
#!/bin/bash

# Installing packages
echo "#########################################################"
echo "Installing Packages"
echo "#########################################################"
sudo yum install wget unzip httpd -y > /dev/null
echo

# Start & Enable Service
echo "#########################################################"
echo "Start & Enable HTTPD service"
echo "#########################################################"
sudo systemctl start httpd
sudo systemctl enable httpd
echo

# Creating Temp Directory
echo "#########################################################"
echo "Starting Artifact Deployment"
echo "#########################################################"
mkdir -p /tmp/webfiles
cd /tmp/webfiles
echo

wget https://www.tooplate.com/zip-templates/2136_kool_form_pack.zip > /dev/null
unzip 2136_kool_form_pack.zip > /dev/null
cp -r 2136_kool_form_pack/* /var/www/html/
echo

# Bounce Service
echo "#########################################################"
echo "Restarting HTTPD Service"
echo "#########################################################"
systemctl restart httpd
echo

# Clean Up
echo "#########################################################"
echo "Removing Temporary Files"
echo "#########################################################"
rm -rf /tmp/webfiles
echo


systemctl status httpd
ls /var/www/html/
```

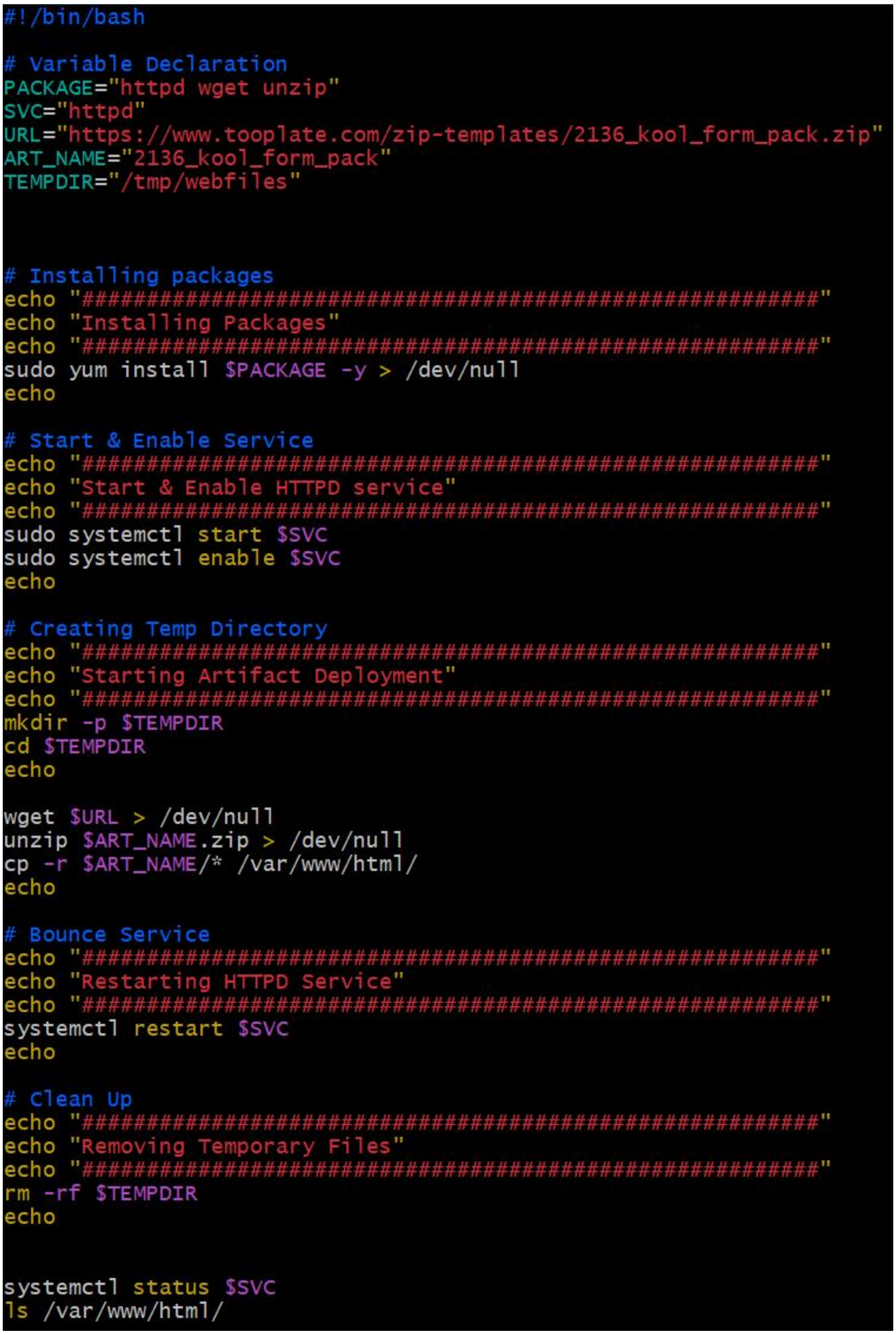
- Variables:



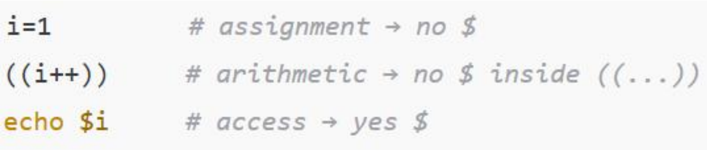
```
[root@scriptbox scripts]# SKILL="DevOps"
[root@scriptbox scripts]# echo SKILL
SKILL
[root@scriptbox scripts]# echo $SKILL
DevOps
[root@scriptbox scripts]# |
```

(without $ it's just a normal string, not a variable)

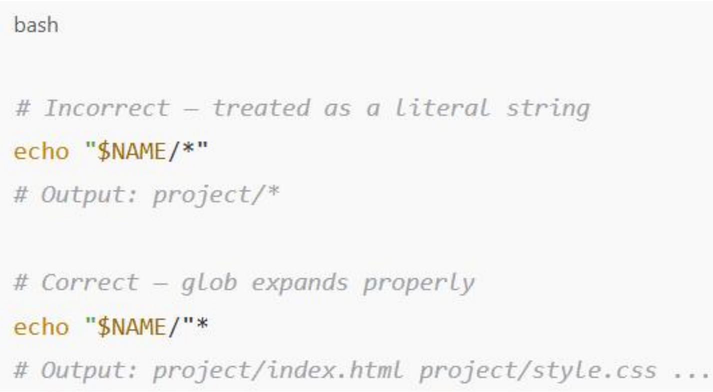
- Using variables:



```
#!/bin/bash

# Variable Declaration
PACKAGE="httpd wget unzip"
SVC="httpd"
URL="https://www.tooplate.com/zip-templates/2136_kool_form_pack.zip"
ART_NAME="2136_kool_form_pack"
TEMPDIR="/tmp/webfiles"



# Installing packages
echo "#########################################################"
echo "Installing Packages"
echo "#########################################################"
sudo yum install $PACKAGE -y > /dev/null
echo

# Start & Enable Service
echo "#########################################################"
echo "Start & Enable HTTPD service"
echo "#########################################################"
sudo systemctl start $SVC
sudo systemctl enable $SVC
echo

# Creating Temp Directory
echo "#########################################################"
echo "Starting Artifact Deployment"
echo "#########################################################"
mkdir -p $TEMPDIR
cd $TEMPDIR
echo

wget $URL > /dev/null
unzip $ART_NAME.zip > /dev/null
cp -r $ART_NAME/* /var/www/html/
echo

# Bounce Service
echo "#########################################################"
echo "Restarting HTTPD Service"
echo "#########################################################"
systemctl restart $SVC
echo

# Clean Up
echo "#########################################################"
echo "Removing Temporary Files"
echo "#########################################################"
rm -rf $TEMPDIR
echo


systemctl status $SVC
ls /var/www/html/
```

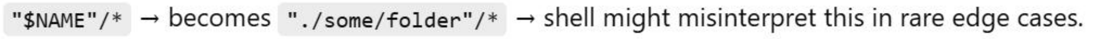
- ✅ Short Rule:
  - Use $i when you're accessing the value of the variable.
  - Use just i when you're declaring, assigning, or iterating over it.
  - ❗ Use $ whenever you would expect to "see the value" — like echo, math comparisons, etc.
  - No $ when you’re giving it a value or defining the loop variable.
  - 

```
i=1          # assignment → no $
((i++))      # arithmetic → no $ inside ((...))
echo $i      # access → yes $
```

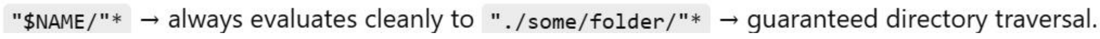
- 

```bash
# Incorrect — treated as a literal string
echo "$NAME/*"
# Output: project/*

# Correct — glob expands properly
echo "$NAME/"*
# Output: project/index.html project/style.css ...
```

  - Don't quote the * if you want shell expansion.
  - Quote the variable part only, like this: "$NAME/"*

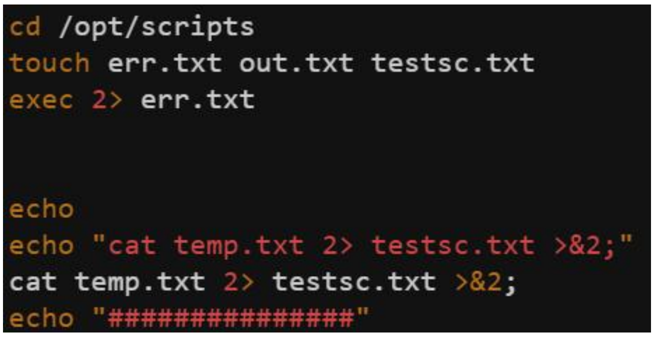


`"$NAME"/*` → becomes `"./some/folder"/*` → shell might misinterpret this in rare edge cases.

  - 

`"$NAME/"*` → always evaluates cleanly to `"./some/folder/"*` → guaranteed directory traversal.

  - 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:***
  - 

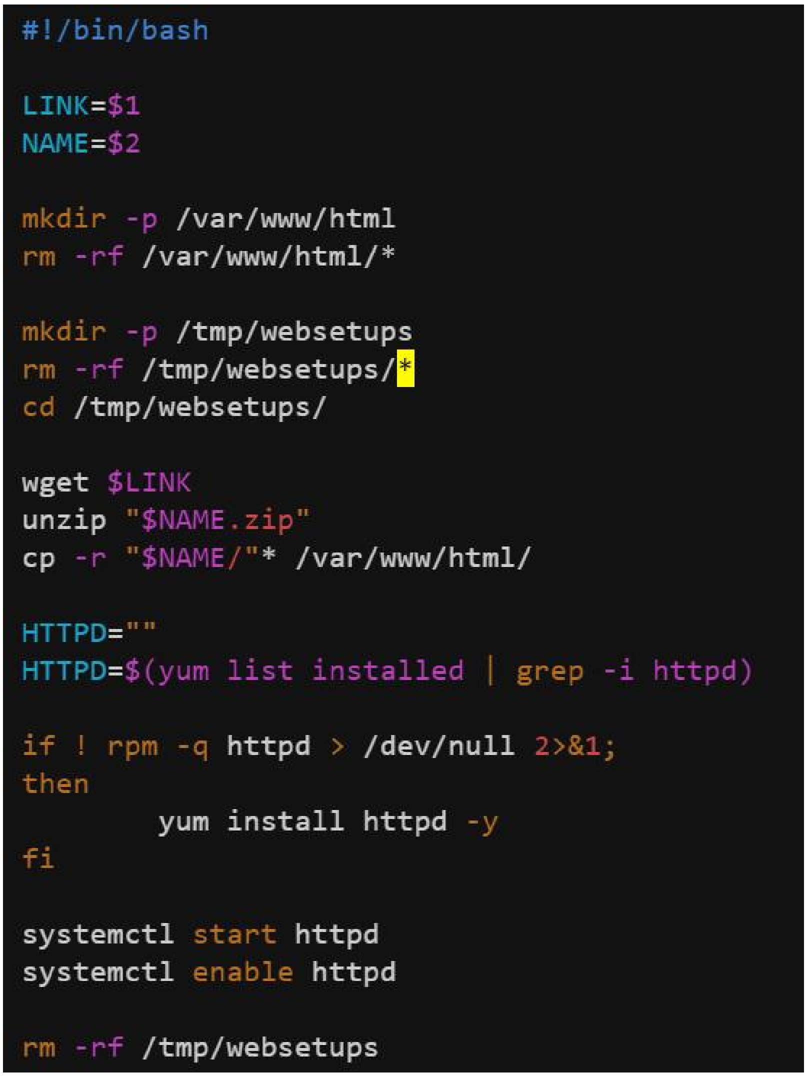
```
cd /opt/scripts
touch err.txt out.txt testsc.txt
exec 2> err.txt


echo
echo "cat temp.txt 2> testsc.txt >&2;"
cat temp.txt 2> testsc.txt >&2;
echo "################"
```

  - 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**)*

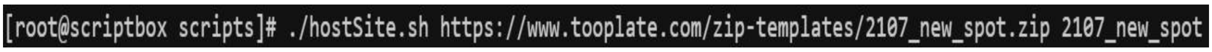
- So, now let's see the above screenshot.
    - I have set all the future ***stderr*** path to ***err.txt***
    - Then I wrote ***cat temp.txt 2> testsc.txt >&2***
    - Here, ***2> testsc.txt*** will override the ***err.txt*** for this command, so for this command, ***stderr*** is referring to ***testsc.txt***.
    - 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.



```
#!/bin/bash

LINK=$1
NAME=$2

mkdir -p /var/www/html
rm -rf /var/www/html/*

mkdir -p /tmp/websetups
rm -rf /tmp/websetups/*
cd /tmp/websetups/

wget $LINK
unzip "$NAME.zip"
cp -r "$NAME/"* /var/www/html/

HTTPD=""
HTTPD=$(yum list installed | grep -i httpd)

if ! rpm -q httpd > /dev/null 2>&1;
then
        yum install httpd -y
fi

systemctl start httpd
systemctl enable httpd

rm -rf /tmp/websetups
```

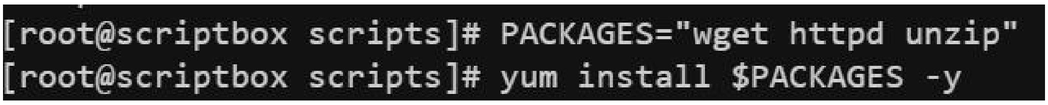
➢ (script to host any site in httpd)

- 

  ```
  [root@scriptbox scripts]# ./hostSite.sh https://www.tooplate.com/zip-templates/2107_new_spot.zip 2107_new_spot
  ```

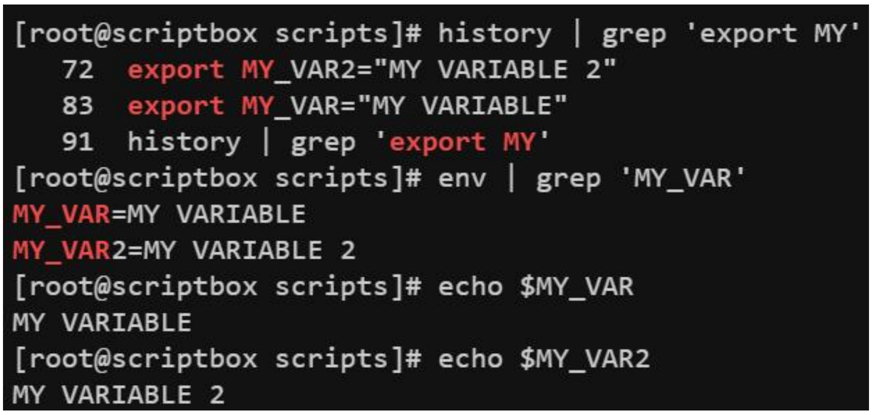
- ***$1 $2*** represents the command line arguments . $0 represents the command itself.

➢ Variables

- 

  ```
  [root@scriptbox scripts]# PACKAGES="wget httpd unzip"
  [root@scriptbox scripts]# yum install $PACKAGES -y
  ```

  - 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)

- 

```
[root@scriptbox scripts]# history | grep 'export MY'
   72  export MY_VAR2="MY VARIABLE 2"
   83  export MY_VAR="MY VARIABLE"
   91  history | grep 'export MY'
[root@scriptbox scripts]# env | grep 'MY_VAR'
MY_VAR=MY VARIABLE
MY_VAR2=MY VARIABLE 2
[root@scriptbox scripts]# echo $MY_VAR
MY VARIABLE
[root@scriptbox scripts]# echo $MY_VAR2
MY VARIABLE 2
```

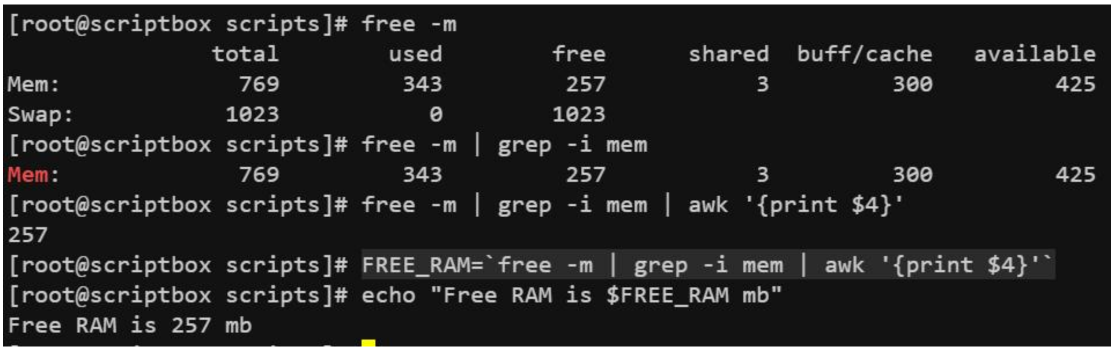
  - ***export*** command is used to set any ***env*** variable.
  - You can check all the ***env*** variables using the commsnd "***env***".

- **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

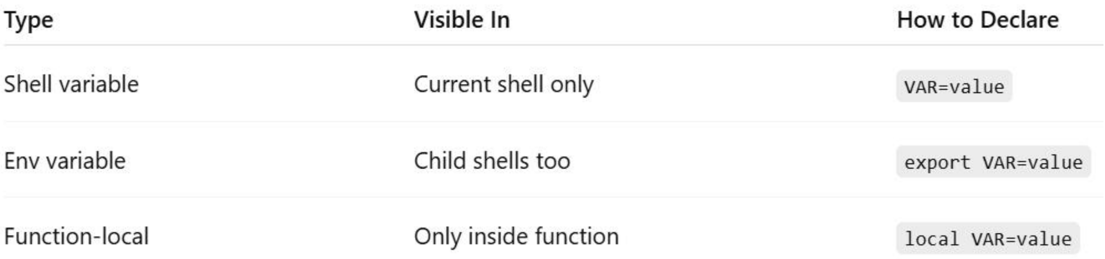
Variable	Description
$0	Script name
$1…$9	First to ninth argument to script
$#	Number of arguments
$@	All arguments as separate quoted strings
$*	All arguments as one word
$?	Exit status of last command
$$	PID of the current shell
$!	PID of last background command

- Command substitution:
  - Stores the output of a command in a variable
  - Use back-tick `` or *$()*



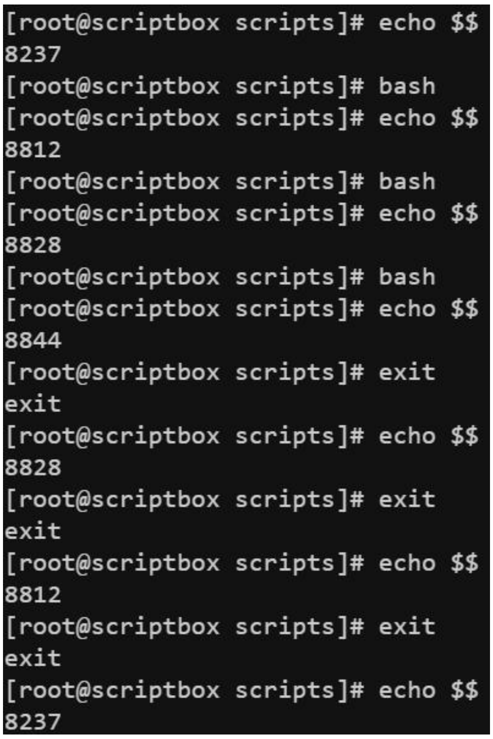
```
[root@scriptbox scripts]# free -m
              total        used        free      shared  buff/cache   available
Mem:            769         343         257           3         300         425
Swap:          1023           0        1023
[root@scriptbox scripts]# free -m | grep -i mem
Mem:            769         343         257           3         300         425
[root@scriptbox scripts]# free -m | grep -i mem | awk '{print $4}'
257
[root@scriptbox scripts]# FREE_RAM=`free -m | grep -i mem | awk '{print $4}'`
[root@scriptbox scripts]# echo "Free RAM is $FREE_RAM mb"
Free RAM is 257 mb
```

  - **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.**



Type	Visible In	How to Declare
Shell variable	Current shell only	`VAR=value`
Env variable	Child shells too	`export VAR=value`
Function-local	Only inside function	`local VAR=value`

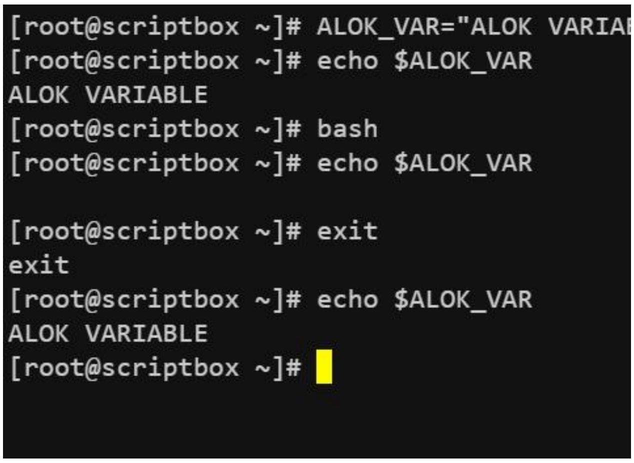
- Child shells:



```
[root@scriptbox scripts]# echo $$
8237
[root@scriptbox scripts]# bash
[root@scriptbox scripts]# echo $$
8812
[root@scriptbox scripts]# bash
[root@scriptbox scripts]# echo $$
8828
[root@scriptbox scripts]# bash
[root@scriptbox scripts]# echo $$
8844
[root@scriptbox scripts]# exit
exit
[root@scriptbox scripts]# echo $$
8828
[root@scriptbox scripts]# exit
exit
[root@scriptbox scripts]# echo $$
8812
[root@scriptbox scripts]# exit
exit
[root@scriptbox scripts]# echo $$
8237
```

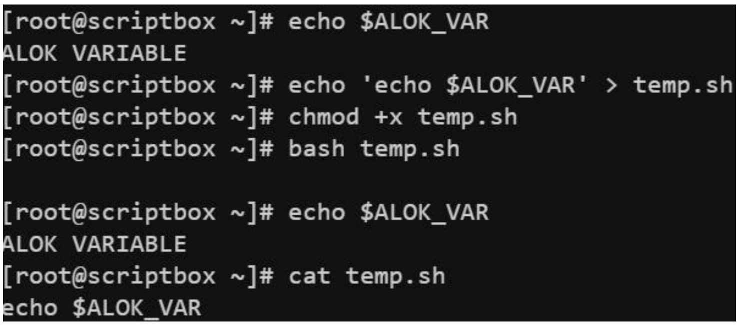
  - *$$* is used to print PID of current shell. 8237(parent shell) -> 8812 -> 8828 -> 8844

- So 3 levels of hierarchy got established.
- You can get out of ***child*** shell to ***parent*** shell using ***exit*** command.



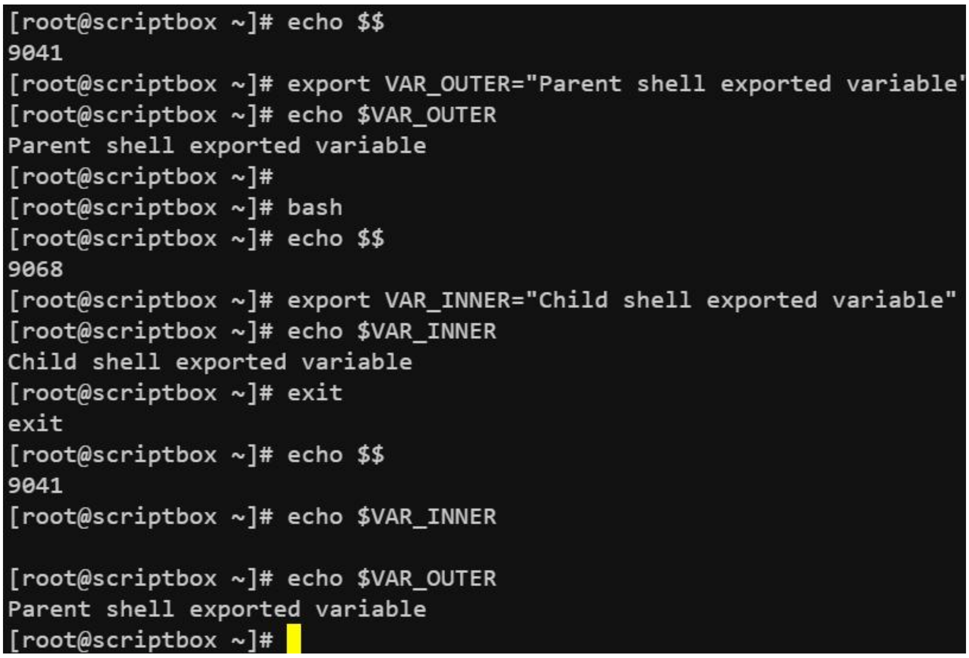
```
[root@scriptbox ~]# ALOK_VAR="ALOK VARIA
[root@scriptbox ~]# echo $ALOK_VAR
ALOK VARIABLE
[root@scriptbox ~]# bash
[root@scriptbox ~]# echo $ALOK_VAR

[root@scriptbox ~]# exit
exit
[root@scriptbox ~]# echo $ALOK_VAR
ALOK VARIABLE
[root@scriptbox ~]# 
```

- In shell script, child shell can't excess the variables declared in parent shell.



```
[root@scriptbox ~]# echo $ALOK_VAR
ALOK VARIABLE
[root@scriptbox ~]# echo 'echo $ALOK_VAR' > temp.sh
[root@scriptbox ~]# chmod +x temp.sh
[root@scriptbox ~]# bash temp.sh

[root@scriptbox ~]# echo $ALOK_VAR
ALOK VARIABLE
[root@scriptbox ~]# cat temp.sh
echo $ALOK_VAR
```

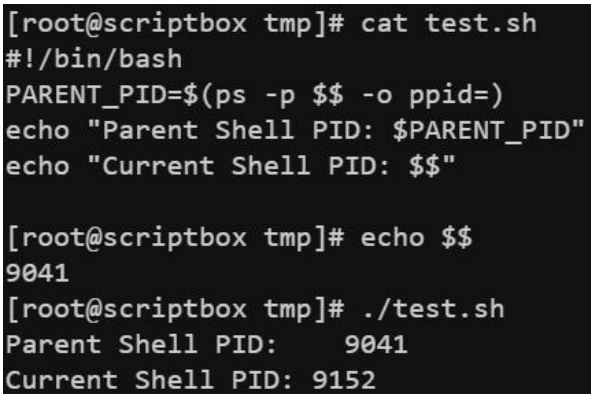
  - By default, shell script files run in child shell. So that also can't access the variables.



```
[root@scriptbox ~]# echo $$
9041
[root@scriptbox ~]# export VAR_OUTER="Parent shell exported variable"
[root@scriptbox ~]# echo $VAR_OUTER
Parent shell exported variable
[root@scriptbox ~]#
[root@scriptbox ~]# bash
[root@scriptbox ~]# echo $$
9068
[root@scriptbox ~]# export VAR_INNER="Child shell exported variable"
[root@scriptbox ~]# echo $VAR_INNER
Child shell exported variable
[root@scriptbox ~]# exit
exit
[root@scriptbox ~]# echo $$
9041
[root@scriptbox ~]# echo $VAR_INNER

[root@scriptbox ~]# echo $VAR_OUTER
Parent shell exported variable
[root@scriptbox ~]# 
```

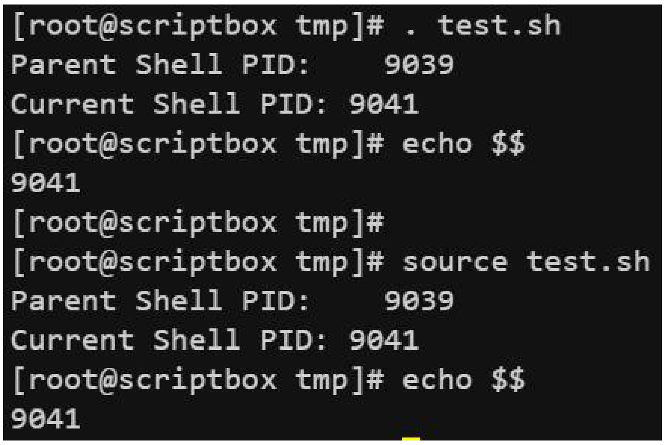
- ***When in a shell, a variable is exported, it'll be available to all of it's child shells but not to parent shell. & If u export a variable and logout and again login, the variable will not be there.***

➢ When u run a script, it by default gets run *in a* child shell.



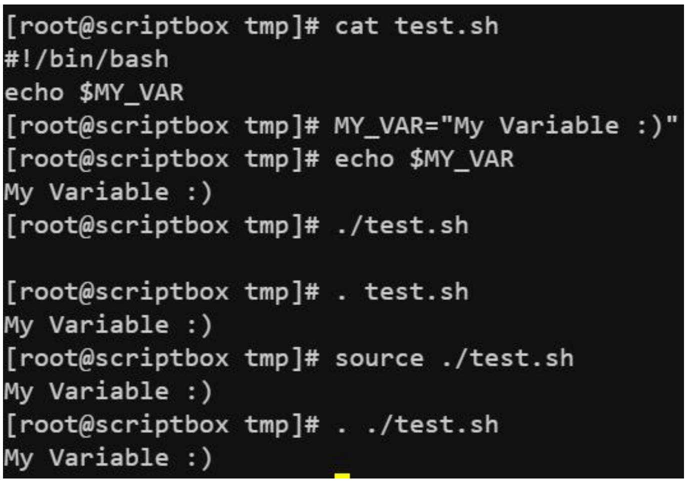
```
[root@scriptbox tmp]# cat test.sh
#!/bin/bash
PARENT_PID=$(ps -p $$ -o ppid=)
echo "Parent Shell PID: $PARENT_PID"
echo "Current Shell PID: $$"

[root@scriptbox tmp]# echo $$
9041
[root@scriptbox tmp]# ./test.sh
Parent Shell PID:    9041
Current Shell PID: 9152
```

(Parent shell PID: 9041, script ran in 9152 which is a child shell)



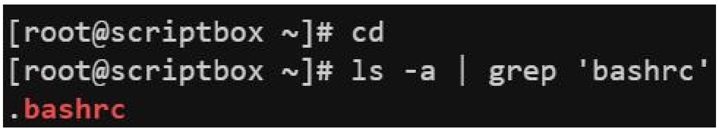
```
[root@scriptbox tmp]# . test.sh
Parent Shell PID:    9039
Current Shell PID: 9041
[root@scriptbox tmp]# echo $$
9041
[root@scriptbox tmp]#
[root@scriptbox tmp]# source test.sh
Parent Shell PID:    9039
Current Shell PID: 9041
[root@scriptbox tmp]# echo $$
9041
```

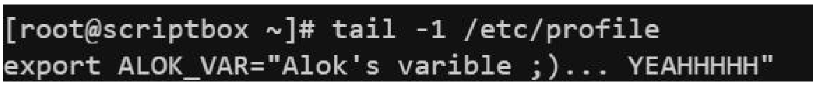
- If u run the script using **.** *(dot<space><filename>)* or ***source*** command, then it'll be run in the current shell only.

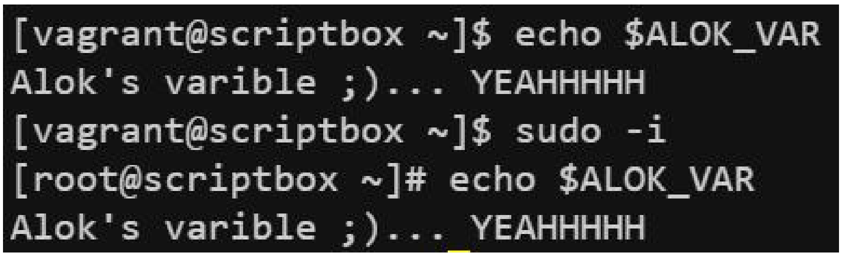


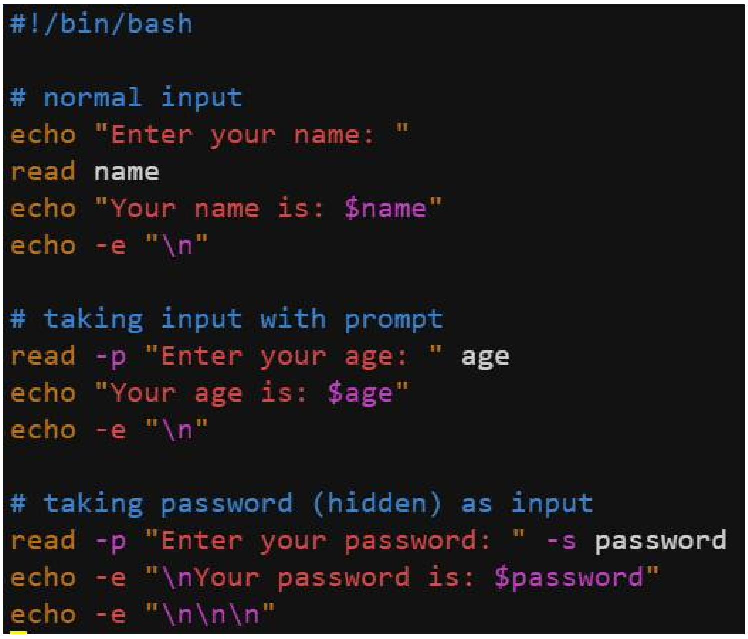
```
[root@scriptbox tmp]# cat test.sh
#!/bin/bash
echo $MY_VAR
[root@scriptbox tmp]# MY_VAR="My Variable :)"
[root@scriptbox tmp]# echo $MY_VAR
My Variable :)
[root@scriptbox tmp]# ./test.sh

[root@scriptbox tmp]# . test.sh
My Variable :)
[root@scriptbox tmp]# source ./test.sh
My Variable :)
[root@scriptbox tmp]# . ./test.sh
My Variable :)
```

(When we ran the script using ***source*** or **.** it accessed the variable MY_VAR)

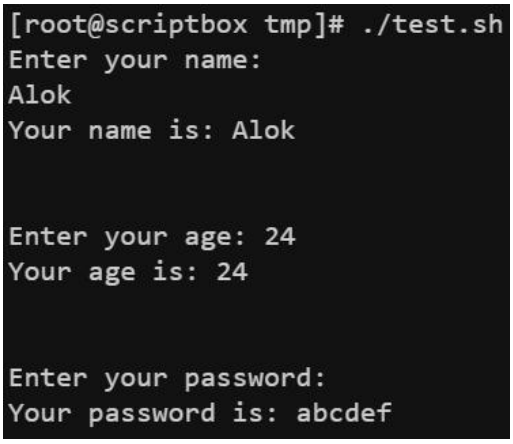
- In home directory of every user(root or any other user) there is an ***.bashrc*** file which is loaded (executed) after log in with that user's shell. If u want to make a variable be accessed for that user even after logging out and logging in, you can ***export*** that variable inside that file.
  - 

    ```
    [root@scriptbox ~]# cd
    [root@scriptbox ~]# ls -a | grep 'bashrc'
    .bashrc
    ```
  - 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***
  - 

    ```
    [root@scriptbox ~]# tail -1 /etc/profile
    export ALOK_VAR="Alok's varible ;)... YEAHHHHH"
    ```
    (like this)
  - 

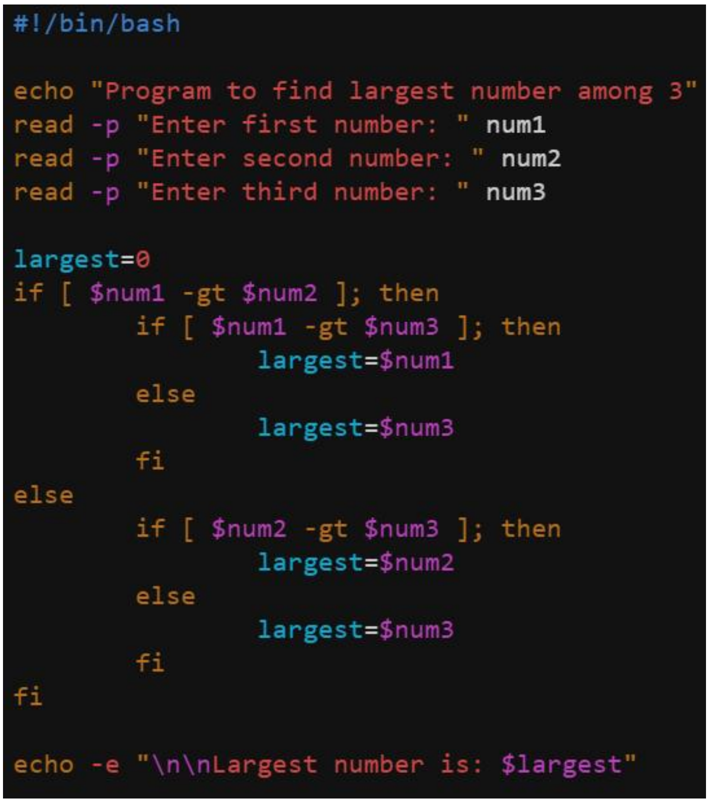
    ```
    [vagrant@scriptbox ~]$ echo $ALOK_VAR
    Alok's varible ;)... YEAHHHHH
    [vagrant@scriptbox ~]$ sudo -i
    [root@scriptbox ~]# echo $ALOK_VAR
    Alok's varible ;)... YEAHHHHH
    ```
    (for all user it is accessible)
  - NOTE: First ***/etc/profile*** file is sourced and then ***.bashrc*** file. So, if same variable is declared in both, then ***.bashrc*** will override that ***/etc/profile*** .
- Taking input from CLI:
  - 

    ```
    #!/bin/bash

    # normal input
    echo "Enter your name: "
    read name
    echo "Your name is: $name"
    echo -e "\n"

    # taking input with prompt
    read -p "Enter your age: " age
    echo "Your age is: $age"
    echo -e "\n"

    # taking password (hidden) as input
    read -p "Enter your password: " -s password
    echo -e "\nYour password is: $password"
    echo -e "\n\n\n"
    ```
    (-p for prompt, -s for hidden input)

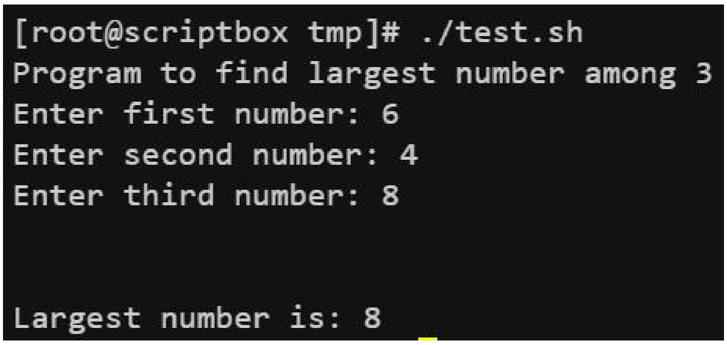


```
[root@scriptbox tmp]# ./test.sh
Enter your name:
Alok
Your name is: Alok


Enter your age: 24
Your age is: 24


Enter your password:
Your password is: abcdef
```

- Decision making(if, elif, else)



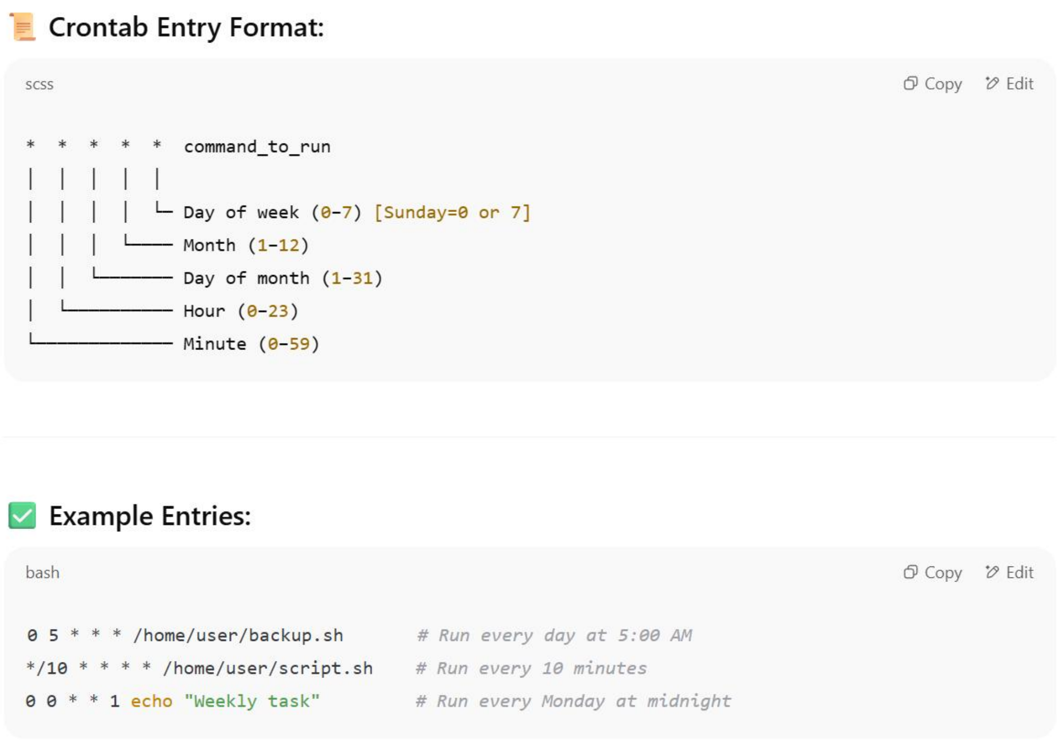
```
#!/bin/bash

echo "Program to find largest number among 3"
read -p "Enter first number: " num1
read -p "Enter second number: " num2
read -p "Enter third number: " num3

largest=0
if [ $num1 -gt $num2 ]; then
        if [ $num1 -gt $num3 ]; then
                largest=$num1
        else
                largest=$num3
        fi
else
        if [ $num2 -gt $num3 ]; then
                largest=$num2
        else
                largest=$num3
        fi
fi

echo -e "\n\nLargest number is: $largest"
```



```
[root@scriptbox tmp]# ./test.sh
Program to find largest number among 3
Enter first number: 6
Enter second number: 4
Enter third number: 8


Largest number is: 8
```

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***.



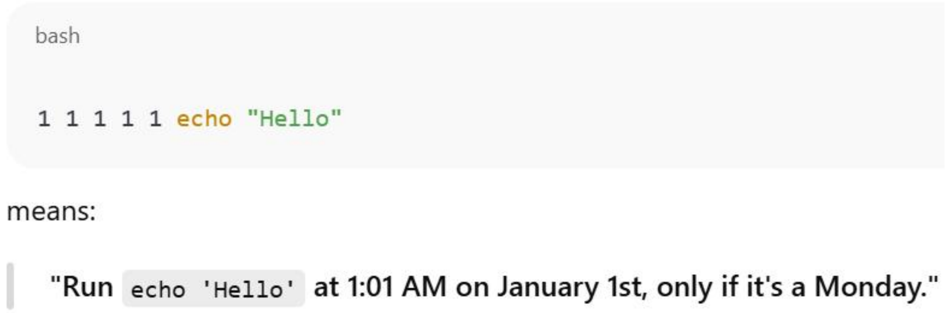
📜 Crontab Entry Format:

```
*  *  *  *  *  command_to_run
|  |  |  |  |
|  |  |  |  └─ Day of week (0-7) [Sunday=0 or 7]
|  |  |  └──── Month (1-12)
|  |  └─────── Day of month (1-31)
|  └────────── Hour (0-23)
└───────────── Minute (0-59)
```

✅ Example Entries:

```bash
0 5 * * * /home/user/backup.sh       # Run every day at 5:00 AM
*/10 * * * * /home/user/script.sh    # Run every 10 minutes
0 0 * * 1 echo "Weekly task"         # Run every Monday at midnight
```

  - Example:

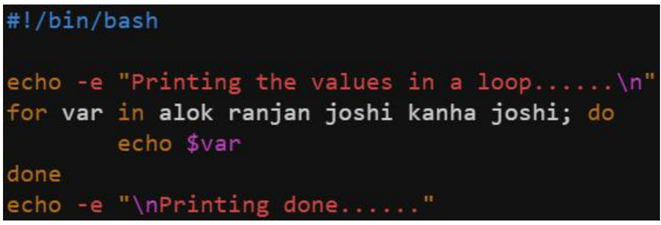


```bash
1 1 1 1 1 echo "Hello"
```

means:

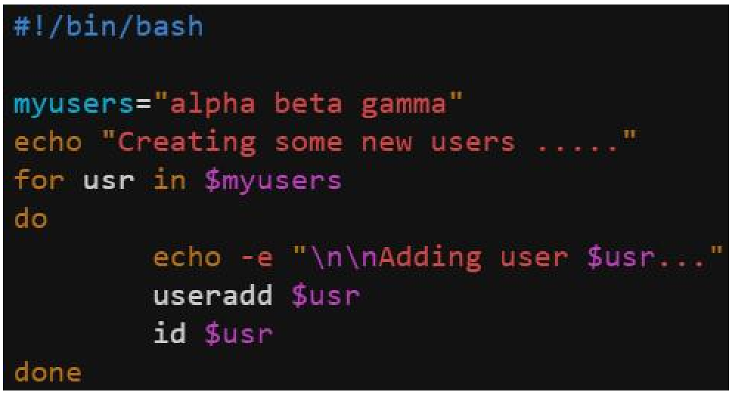
> "Run `echo 'Hello'` at 1:01 AM on January 1st, only if it's a Monday."

- *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.



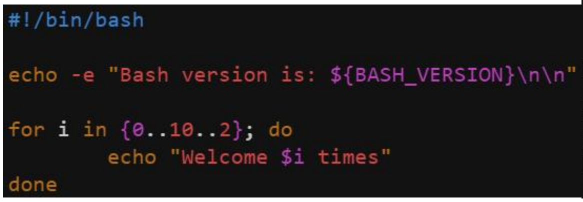
```bash
#!/bin/bash

echo -e "Printing the values in a loop......\n"
for var in alok ranjan joshi kanha joshi; do
        echo $var
done
echo -e "\nPrinting done......"
```

(looping through an array)

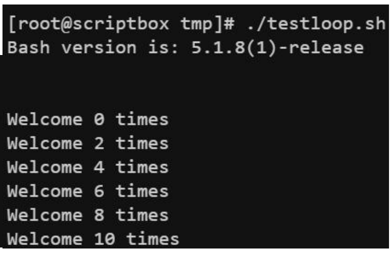


```
#!/bin/bash

myusers="alpha beta gamma"
echo "Creating some new users ....."
for usr in $myusers
do
        echo -e "\n\nAdding user $usr..."
        useradd $usr
        id $usr
done
```

- 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.

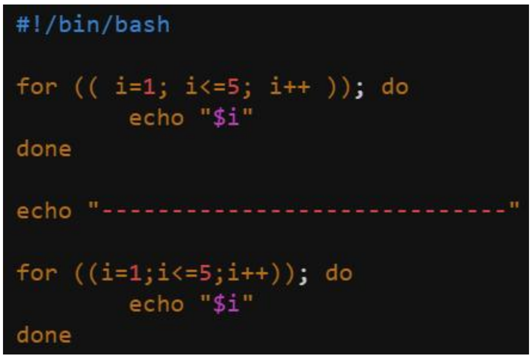


```
#!/bin/bash

echo -e "Bash version is: ${BASH_VERSION}\n\n"

for i in {0..10..2}; do
        echo "Welcome $i times"
done
```

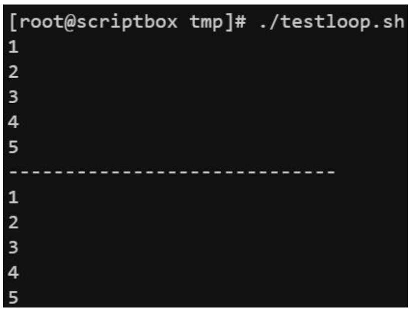


```
[root@scriptbox tmp]# ./testloop.sh
Bash version is: 5.1.8(1)-release


Welcome 0 times
Welcome 2 times
Welcome 4 times
Welcome 6 times
Welcome 8 times
Welcome 10 times
```

- 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)

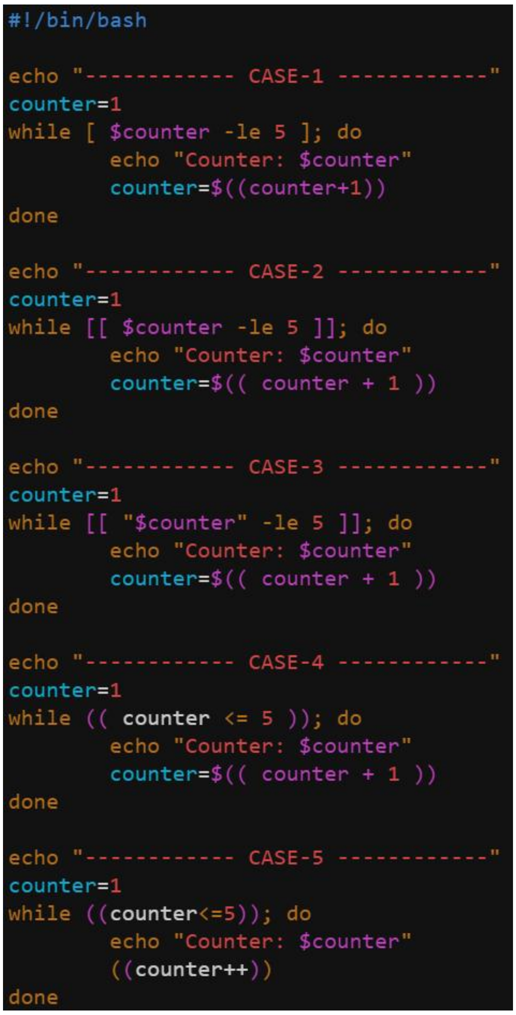


```
#!/bin/bash

for (( i=1; i<=5; i++ )); do
        echo "$i"
done

echo "-----------------------------"

for ((i=1;i<=5;i++)); do
        echo "$i"
done
```

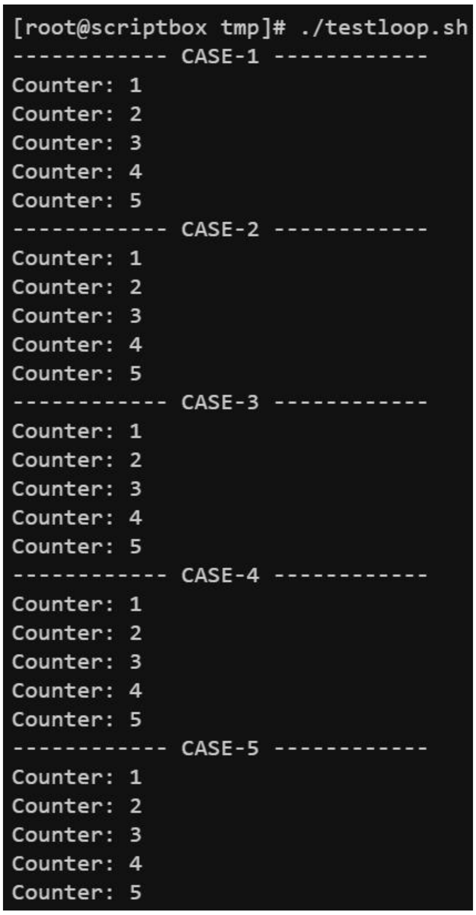


```
[root@scriptbox tmp]# ./testloop.sh
1
2
3
4
5
-----------------------------
1
2
3
4
5
```

- gave to examples to show that space is not required here

- 

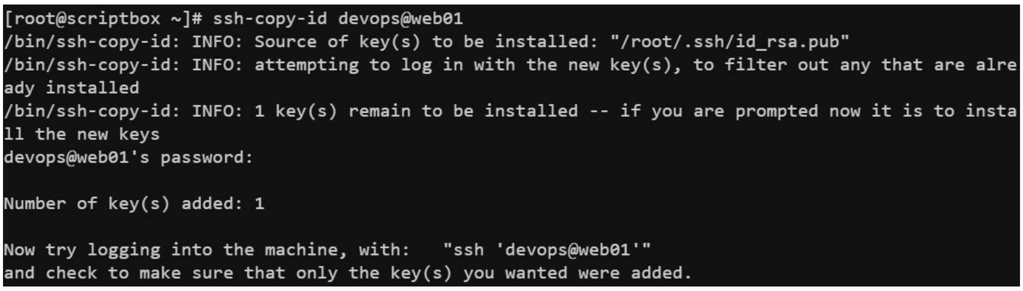
```
#!/bin/bash

echo "------------ CASE-1 ------------"
counter=1
while [ $counter -le 5 ]; do
        echo "Counter: $counter"
        counter=$((counter+1))
done

echo "------------ CASE-2 ------------"
counter=1
while [[ $counter -le 5 ]]; do
        echo "Counter: $counter"
        counter=$(( counter + 1 ))
done

echo "------------ CASE-3 ------------"
counter=1
while [[ "$counter" -le 5 ]]; do
        echo "Counter: $counter"
        counter=$(( counter + 1 ))
done

echo "------------ CASE-4 ------------"
counter=1
while (( counter <= 5 )); do
        echo "Counter: $counter"
        counter=$(( counter + 1 ))
done

echo "------------ CASE-5 ------------"
counter=1
while ((counter<=5)); do
        echo "Counter: $counter"
        ((counter++))
done
```



```
[root@scriptbox tmp]# ./testloop.sh
------------ CASE-1 ------------
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
------------ CASE-2 ------------
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
------------ CASE-3 ------------
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
------------ CASE-4 ------------
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
------------ CASE-5 ------------
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
```

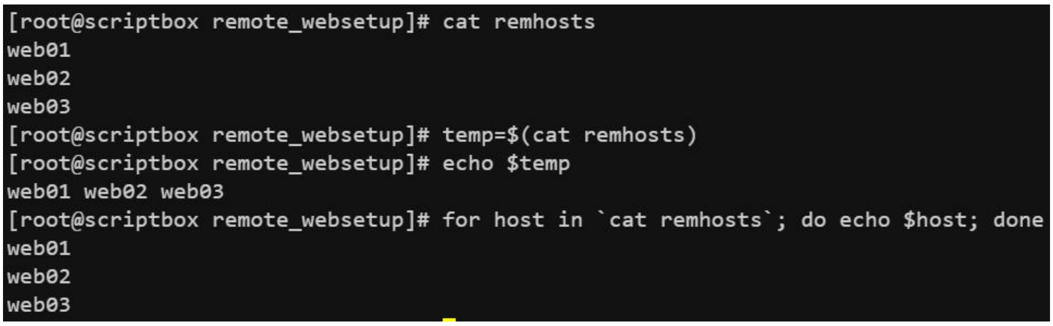
  - Look case-2 & case-3 properly.
  - You can’t write count=count+1 or count=$count+1

- 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. **ssh-keygen** is the command to generate the ssh key.
  - Then **ssh-copy-id devops@web01** (means **ssh-copy-id <username>@<vm name>**)



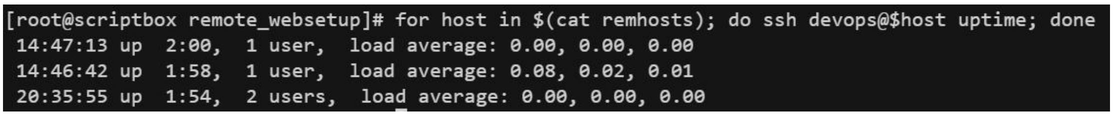
```
[root@scriptbox ~]# ssh-copy-id devops@web01
/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/root/.ssh/id_rsa.pub"
/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are alre
ady installed
/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to insta
ll the new keys
devops@web01's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'devops@web01'"
and check to make sure that only the key(s) you wanted were added.
```

- 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**.



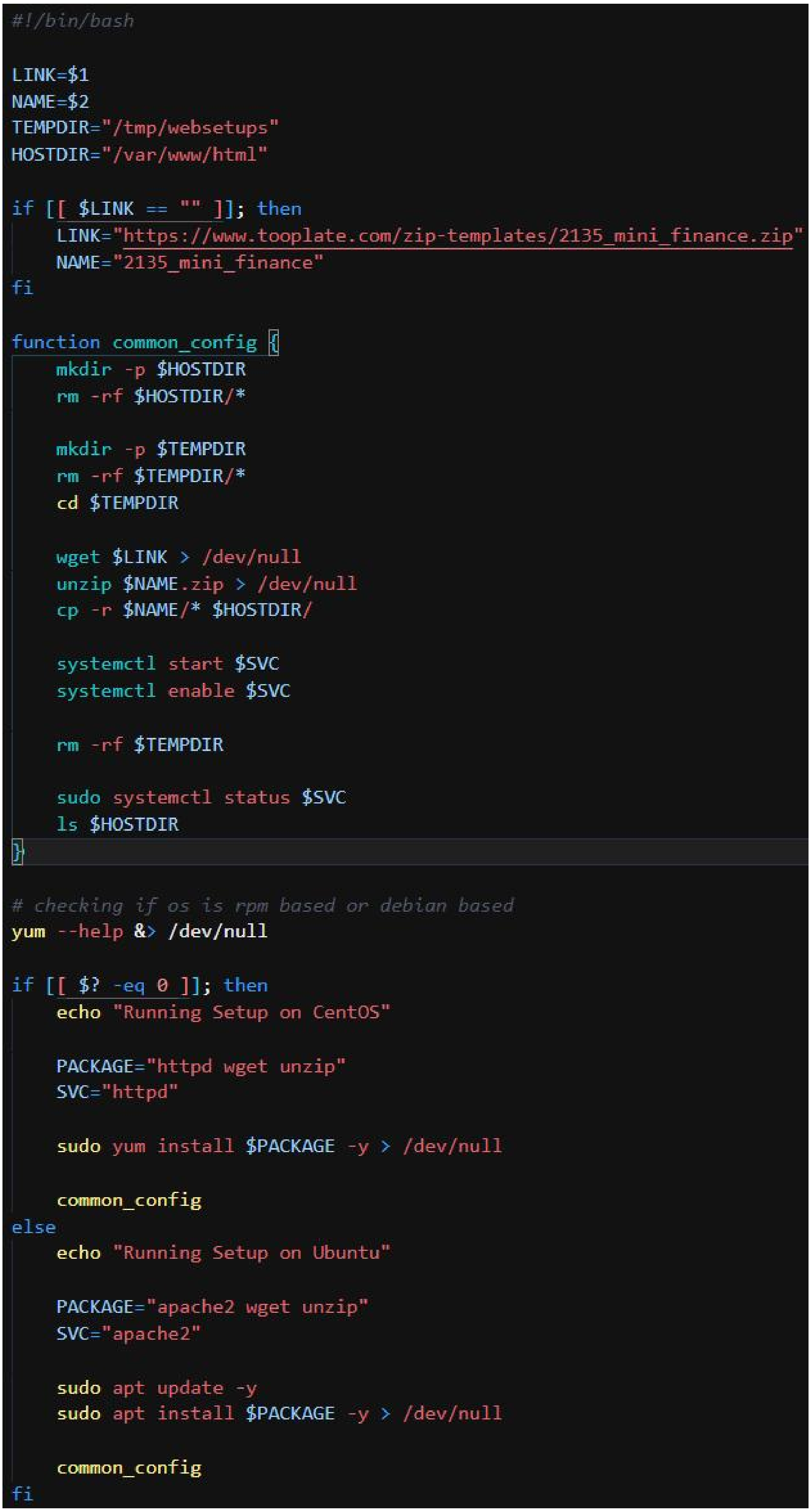
```
[root@scriptbox remote_websetup]# cat remhosts
web01
web02
web03
[root@scriptbox remote_websetup]# temp=$(cat remhosts)
[root@scriptbox remote_websetup]# echo $temp
web01 web02 web03
[root@scriptbox remote_websetup]# for host in `cat remhosts`; do echo $host; done
web01
web02
web03
```

  - 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.

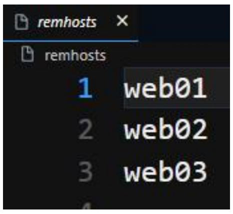


```
[root@scriptbox remote_websetup]# for host in $(cat remhosts); do ssh devops@$host uptime; done
 14:47:13 up  2:00,  1 user,  load average: 0.00, 0.00, 0.00
 14:46:42 up  1:58,  1 user,  load average: 0.08, 0.02, 0.01
 20:35:55 up  1:54,  2 users,  load average: 0.00, 0.00, 0.00
```

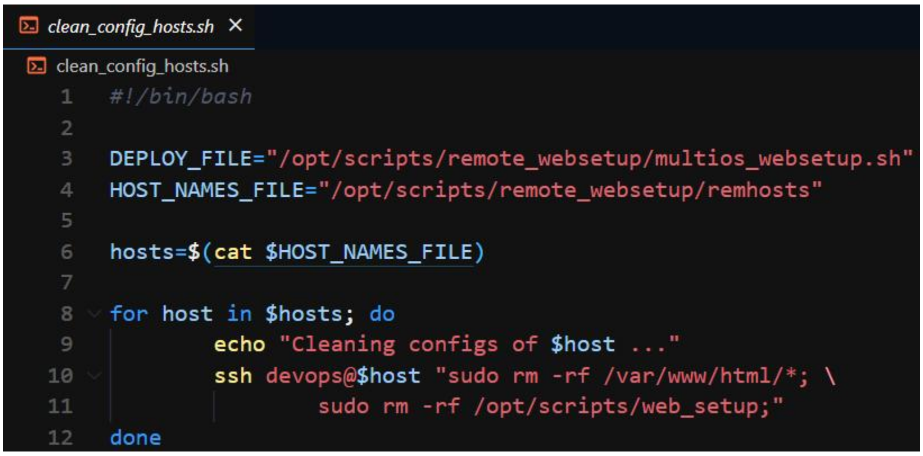
- **scp** command:
  - It is used to **download/upload** (basically **copy**) the files between **current & remote** machines.
  - It uses **ssh** protocol.
  - **scp <path from> <path to>**
    - **scp note.txt alok@web01:home/tmp/** (upload)
    - **scp alok@web01:home/tmp/note.txt ./note.txt** (download)



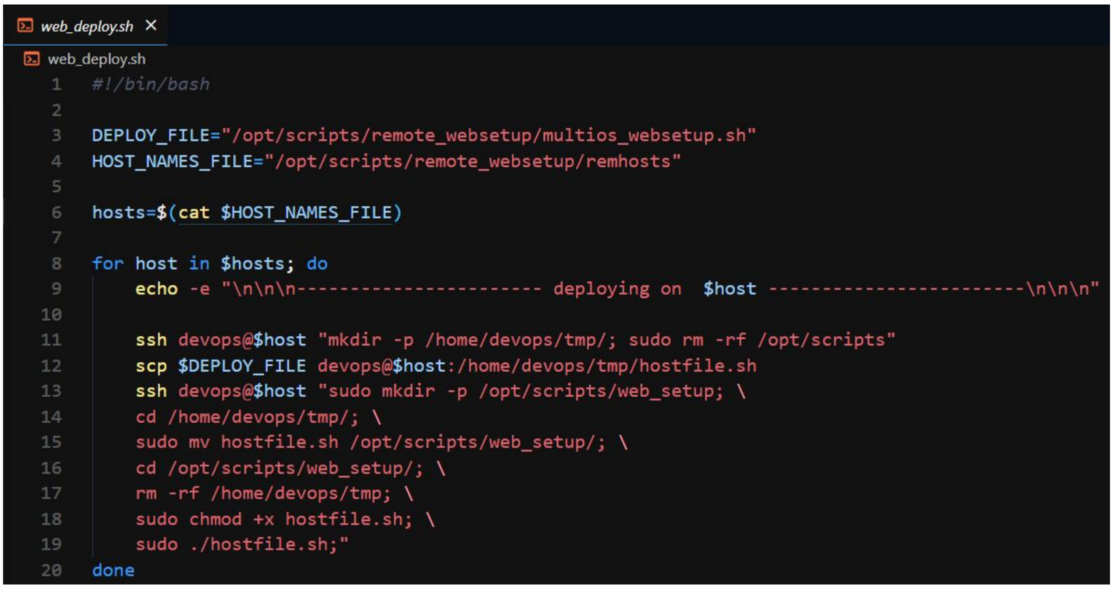
```bash
#!/bin/bash

LINK=$1
NAME=$2
TEMPDIR="/tmp/websetups"
HOSTDIR="/var/www/html"

if [[ $LINK == "" ]]; then
    LINK="https://www.tooplate.com/zip-templates/2135_mini_finance.zip"
    NAME="2135_mini_finance"
fi

function common_config {
    mkdir -p $HOSTDIR
    rm -rf $HOSTDIR/*

    mkdir -p $TEMPDIR
    rm -rf $TEMPDIR/*
    cd $TEMPDIR

    wget $LINK > /dev/null
    unzip $NAME.zip > /dev/null
    cp -r $NAME/* $HOSTDIR/

    systemctl start $SVC
    systemctl enable $SVC

    rm -rf $TEMPDIR

    sudo systemctl status $SVC
    ls $HOSTDIR
}

# checking if os is rpm based or debian based
yum --help &> /dev/null

if [[ $? -eq 0 ]]; then
    echo "Running Setup on CentOS"

    PACKAGE="httpd wget unzip"
    SVC="httpd"

    sudo yum install $PACKAGE -y > /dev/null

    common_config
else
    echo "Running Setup on Ubuntu"

    PACKAGE="apache2 wget unzip"
    SVC="apache2"

    sudo apt update -y
    sudo apt install $PACKAGE -y > /dev/null

    common_config
fi
```

- 

```
web01
web02
web03
```

- 

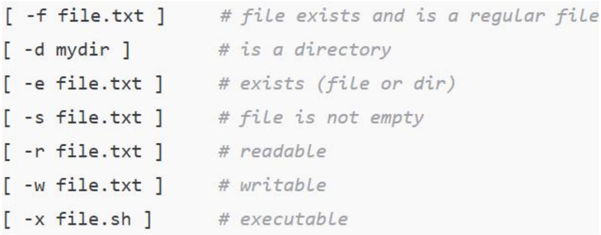
```bash
#!/bin/bash

DEPLOY_FILE="/opt/scripts/remote_websetup/multios_websetup.sh"
HOST_NAMES_FILE="/opt/scripts/remote_websetup/remhosts"

hosts=$(cat $HOST_NAMES_FILE)

for host in $hosts; do
        echo "Cleaning configs of $host ..."
        ssh devops@$host "sudo rm -rf /var/www/html/*; \
                sudo rm -rf /opt/scripts/web_setup;"
done
```

- 

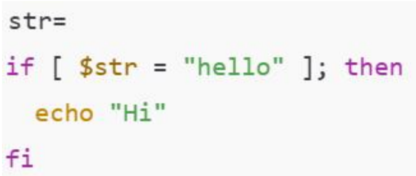
```bash
#!/bin/bash

DEPLOY_FILE="/opt/scripts/remote_websetup/multios_websetup.sh"
HOST_NAMES_FILE="/opt/scripts/remote_websetup/remhosts"

hosts=$(cat $HOST_NAMES_FILE)

for host in $hosts; do
    echo -e "\n\n\n---------------------- deploying on  $host -----------------------\n\n\n"

    ssh devops@$host "mkdir -p /home/devops/tmp/; sudo rm -rf /opt/scripts"
    scp $DEPLOY_FILE devops@$host:/home/devops/tmp/hostfile.sh
    ssh devops@$host "sudo mkdir -p /opt/scripts/web_setup; \
    cd /home/devops/tmp/; \
    sudo mv hostfile.sh /opt/scripts/web_setup/; \
    cd /opt/scripts/web_setup/; \
    rm -rf /home/devops/tmp; \
    sudo chmod +x hostfile.sh; \
    sudo ./hostfile.sh;"
done
```

- 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:*

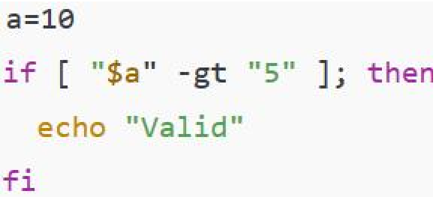
- Use of **$** :
  - Where you are accessing a variable like **echo $my_var**
  - Inside string i.e. **"/tmp/$dir_name/"** (here dir_name is a variable, written inside a *double quote*)
  - In arithmetic expression like **sum = $((x + y))** here *$* means return the output of the expression so that *sum* 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.
  - 

```
[ -f file.txt ]     # file exists and is a regular file
[ -d mydir ]        # is a directory
[ -e file.txt ]     # exists (file or dir)
[ -s file.txt ]     # file is not empty
[ -r file.txt ]     # readable
[ -w file.txt ]     # writable
[ -x file.sh ]      # executable
```

    (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 [..]
  - 

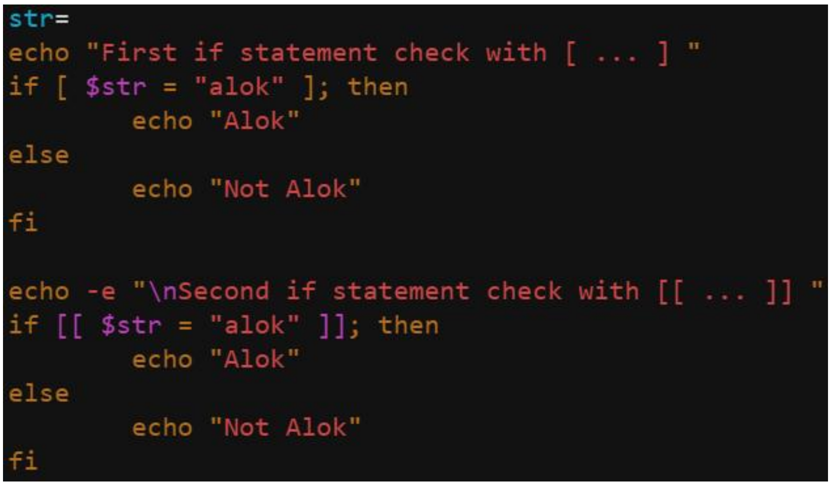
```
str=
if [ $str = "hello" ]; then
  echo "Hi"
fi
```

    - Here, it'll become [= "hello"], so will give error Error: unary operator expected.
    - So, best practice is **["$str" = "hello"]**

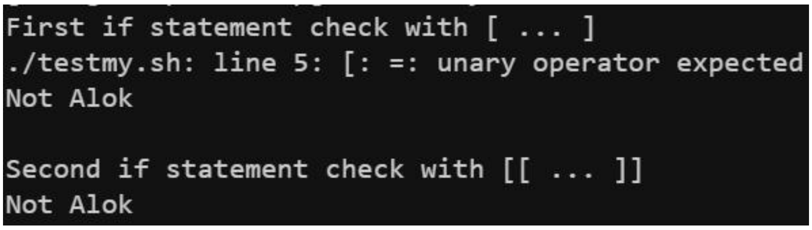
- 

```
a=10
if [ "$a" -gt "5" ]; then
  echo "Valid"
fi
```

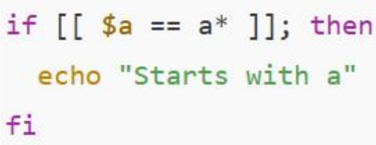
    - as we know ***-gt*** is for numeric values not for strings. But here ***a*** is a number and this string represents a number. So here ***-gt*** will work fine.
    - If ***a="alok"*** and we have written **if ["$a" -eq "alok"]** it'll give error as ***-eq*** is not for strings.

- 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.
    - 

```
str=
echo "First if statement check with [ ... ] "
if [ $str = "alok" ]; then
        echo "Alok"
else
        echo "Not Alok"
fi

echo -e "\nSecond if statement check with [[ ... ]] "
if [[ $str = "alok" ]]; then
        echo "Alok"
else
        echo "Not Alok"
fi
```

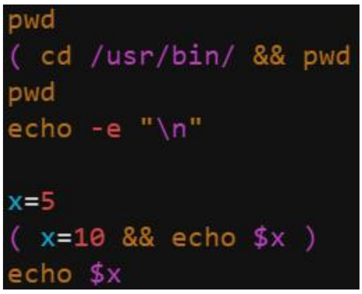
    - 

```
First if statement check with [ ... ]
./testmy.sh: line 5: [: =: unary operator expected
Not Alok

Second if statement check with [[ ... ]]
Not Alok
```

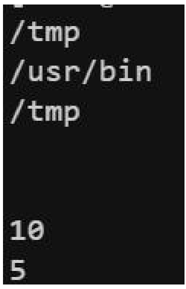
  - One additional thing is ***regex matching*** & ***pattern matching***.
    - 

```
if [[ $a == a* ]]; then
  echo "Starts with a"
fi
```

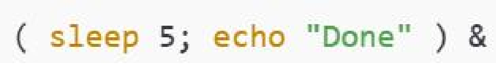
  - In case of **[…]** we were not able to use **&&** and **| |** but in case of **[[…]]** we can use **&&** and **| |** inside that.
    - **if [[$x -gt 5 && $x -lt 20]]**

- 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.
    - 

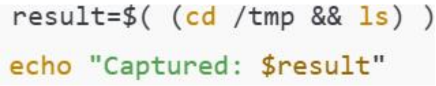
      ```
      pwd
      ( cd /usr/bin/ && pwd
      pwd
      echo -e "\n"

      x=5
      ( x=10 && echo $x )
      echo $x
      ```

      (code)
    - 

      ```
      /tmp
      /usr/bin
      /tmp


      10
      5
      ```

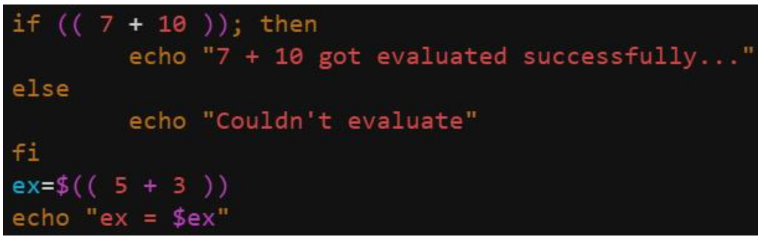
      (output)
  - We can group commands with redirection.
    - ***(echo "Line 1"; echo "Line 2") > output.txt***
  - Can be used in pipelines
    - ***(cd /tmp && ls) | grep config***
  - 

    ```
    ( sleep 5; echo "Done" ) &
    ```

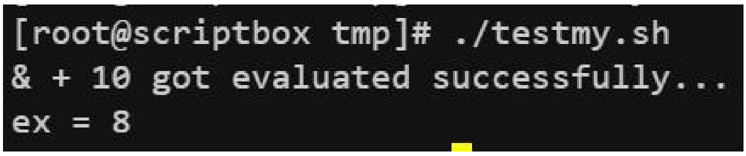
    - This **&** at the end means it run these grouped commands in the ***background***.
  - 

    ```
    result=$( (cd /tmp && ls) )
    echo "Captured: $result"
    ```

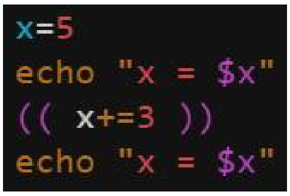
    (it is not same as $((cd /tmp && ls))
  - **(echo "One"; (echo "Two")) | grep T**
    - You might thinking here, echo "Two" will return "Two" so ultimately:
      - ***(echo "One"; "Two") | grep T***
        - But it becomes **One Two** at the and so doesn't give error. Output will be "Two"
    - But 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.*
    - 

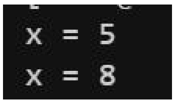
      ```
      if (( 7 + 10 )); then
              echo "7 + 10 got evaluated successfully..."
      else
              echo "Couldn't evaluate"
      fi
      ex=$(( 5 + 3 ))
      echo "ex = $ex"
      ```

      (code)
    - 

      ```
      [root@scriptbox tmp]# ./testmy.sh
      & + 10 got evaluated successfully...
      ex = 8
      ```

      (output)
  - Arithmetic operations
    - 

      ```
      x=5
      echo "x = $x"
      (( x+=3 ))
      echo "x = $x"
      ```

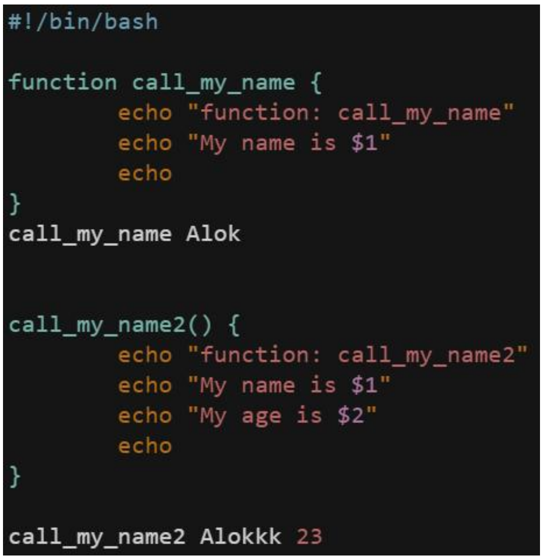
      (code)
    - 

      ```
       x = 5
       x = 8
      ```

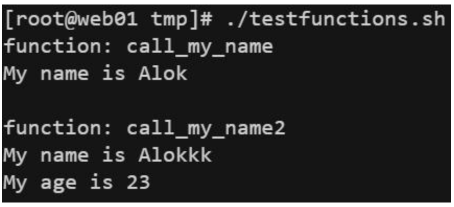
      (output)
- ***For […], [[…]] spaces are required but for (…) and ((…)) space are optional.***

- 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 fine the variable having name **filename.txt**

- Functions in Shell Scripts



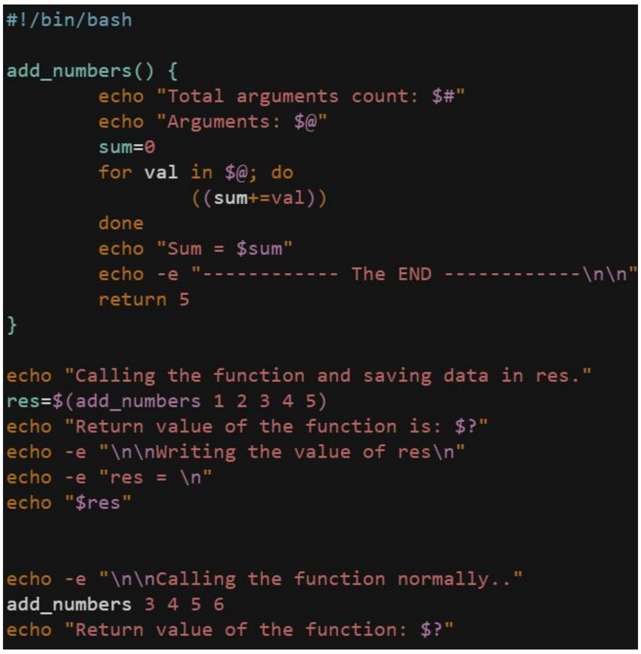
```
#!/bin/bash

function call_my_name {
        echo "function: call_my_name"
        echo "My name is $1"
        echo
}
call_my_name Alok


call_my_name2() {
        echo "function: call_my_name2"
        echo "My name is $1"
        echo "My age is $2"
        echo
}

call_my_name2 Alokkk 23
```



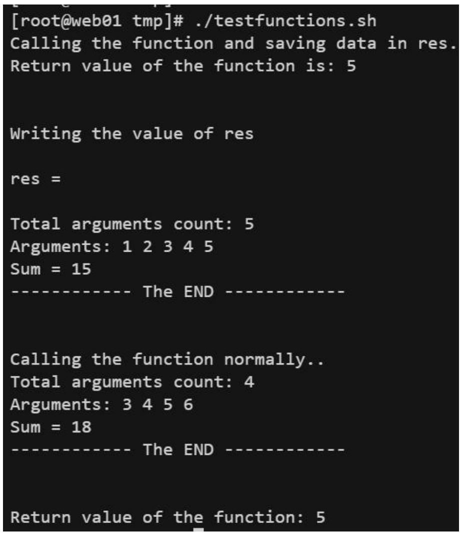
```
[root@web01 tmp]# ./testfunctions.sh
function: call_my_name
My name is Alok

function: call_my_name2
My name is Alokkk
My age is 23
```

- 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**.
  -
  -



```
#!/bin/bash

add_numbers() {
        echo "Total arguments count: $#"
        echo "Arguments: $@"
        sum=0
        for val in $@; do
                ((sum+=val))
        done
        echo "Sum = $sum"
        echo -e "------------ The END ------------\n\n"
        return 5
}

echo "Calling the function and saving data in res."
res=$(add_numbers 1 2 3 4 5)
echo "Return value of the function is: $?"
echo -e "\n\nWriting the value of res\n"
echo -e "res = \n"
echo "$res"


echo -e "\n\nCalling the function normally.."
add_numbers 3 4 5 6
echo "Return value of the function: $?"
```



```
[root@web01 tmp]# ./testfunctions.sh
Calling the function and saving data in res.
Return value of the function is: 5


Writing the value of res

res =

Total arguments count: 5
Arguments: 1 2 3 4 5
Sum = 15
------------ The END ------------


Calling the function normally..
Total arguments count: 4
Arguments: 3 4 5 6
Sum = 18
------------ The END ------------


Return value of the function: 5
```

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