DTL – Bash Scripts

shutterstock.com · 1342796927

Useful Links

Bash Scripting Tutorial for Beginners

https://linuxconfig.org/bash-scripting-tutorial-for-beginners

40 Essential Linux Commands That Every User Should Know

https://www.hostinger.in/tutorials/linux-commands

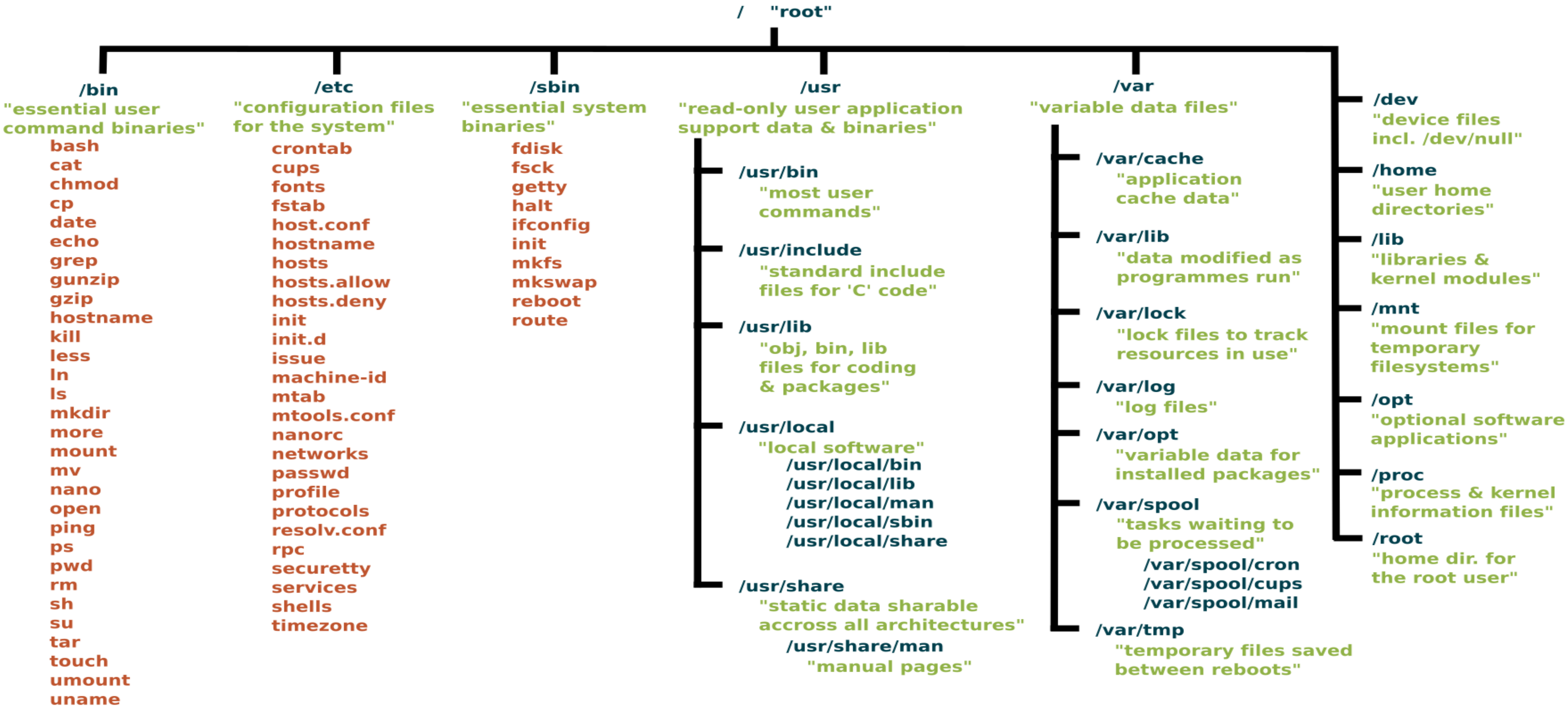
Bash Reference Manual

https://www.gnu.org/software/bash/manual/html_node/

Linux File System

see : https://www.linuxfoundation.org/blog/blog/classic-sysadmin-the-linux-filesystem-explained

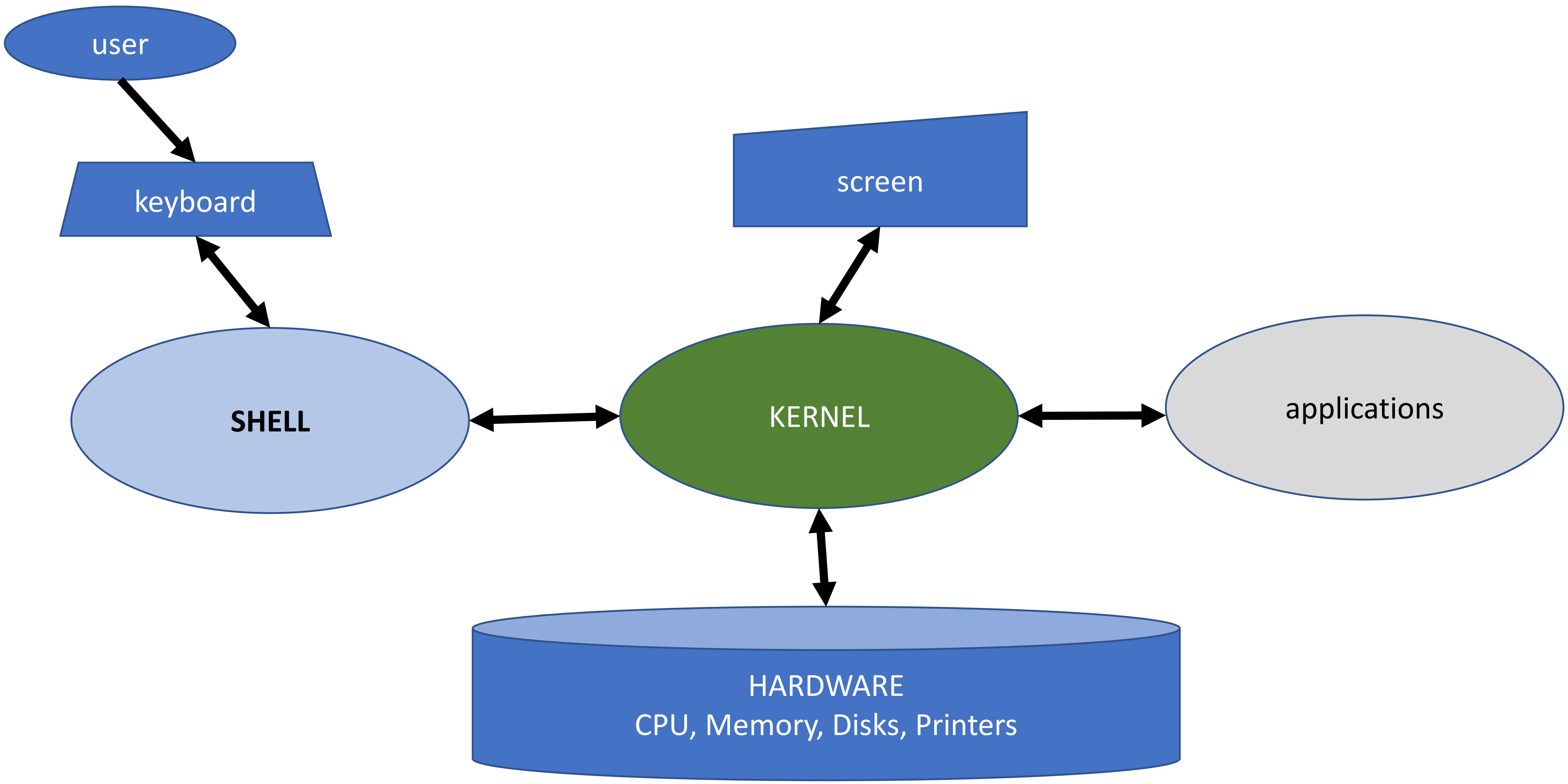
```
#apt install tree
#tree –L 1 /
```



/ "root"

- /bin "essential user command binaries"
  - bash
  - cat
  - chmod
  - cp
  - date
  - echo
  - grep
  - gunzip
  - gzip
  - hostname
  - kill
  - less
  - ln
  - ls
  - mkdir
  - more
  - mount
  - mv
  - nano
  - open
  - ping
  - ps
  - pwd
  - rm
  - sh
  - su
  - tar
  - touch
  - umount
  - uname
- /etc "configuration files for the system"
  - crontab
  - cups
  - fonts
  - fstab
  - host.conf
  - hostname
  - hosts
  - hosts.allow
  - hosts.deny
  - init
  - init.d
  - issue
  - machine-id
  - mtab
  - mtools.conf
  - nanorc
  - networks
  - passwd
  - profile
  - protocols
  - resolv.conf
  - rpc
  - securetty
  - services
  - shells
  - timezone
- /sbin "essential system binaries"
  - fdisk
  - fsck
  - getty
  - halt
  - ifconfig
  - init
  - mkfs
  - mkswap
  - reboot
  - route
- /usr "read-only user application support data & binaries"
  - /usr/bin "most user commands"
  - /usr/include "standard include files for 'C' code"
  - /usr/lib "obj, bin, lib files for coding & packages"
  - /usr/local "local software"
    - /usr/local/bin
    - /usr/local/lib
    - /usr/local/man
    - /usr/local/sbin
    - /usr/local/share
  - /usr/share "static data sharable accross all architectures"
    - /usr/share/man "manual pages"
- /var "variable data files"
  - /var/cache "application cache data"
  - /var/lib "data modified as programmes run"
  - /var/lock "lock files to track resources in use"
  - /var/log "log files"
  - /var/opt "variable data for installed packages"
  - /var/spool "tasks waiting to be processed"
    - /var/spool/cron
    - /var/spool/cups
    - /var/spool/mail
  - /var/tmp "temporary files saved between reboots"
- /dev "device files incl. /dev/null"
- /home "user home directories"
- /lib "libraries & kernel modules"
- /mnt "mount files for temporary filesystems"
- /opt "optional software applications"
- /proc "process & kernel information files"
- /root "home dir. for the root user"

Shell in context!



About ‘bash’ scripting

Bash

Bash is a command language ***interpreter***. It is widely available on various operating systems and is a default command interpreter on most GNU/Linux systems. The name is an acronym for the ‘**B**ourne-**A**gain **SH**ell’.

- Tells the Kernel what programs to use and how to run them

Shell

Shell is a ***macro processor*** which allows for an interactive or non-interactive command execution.

Scripting

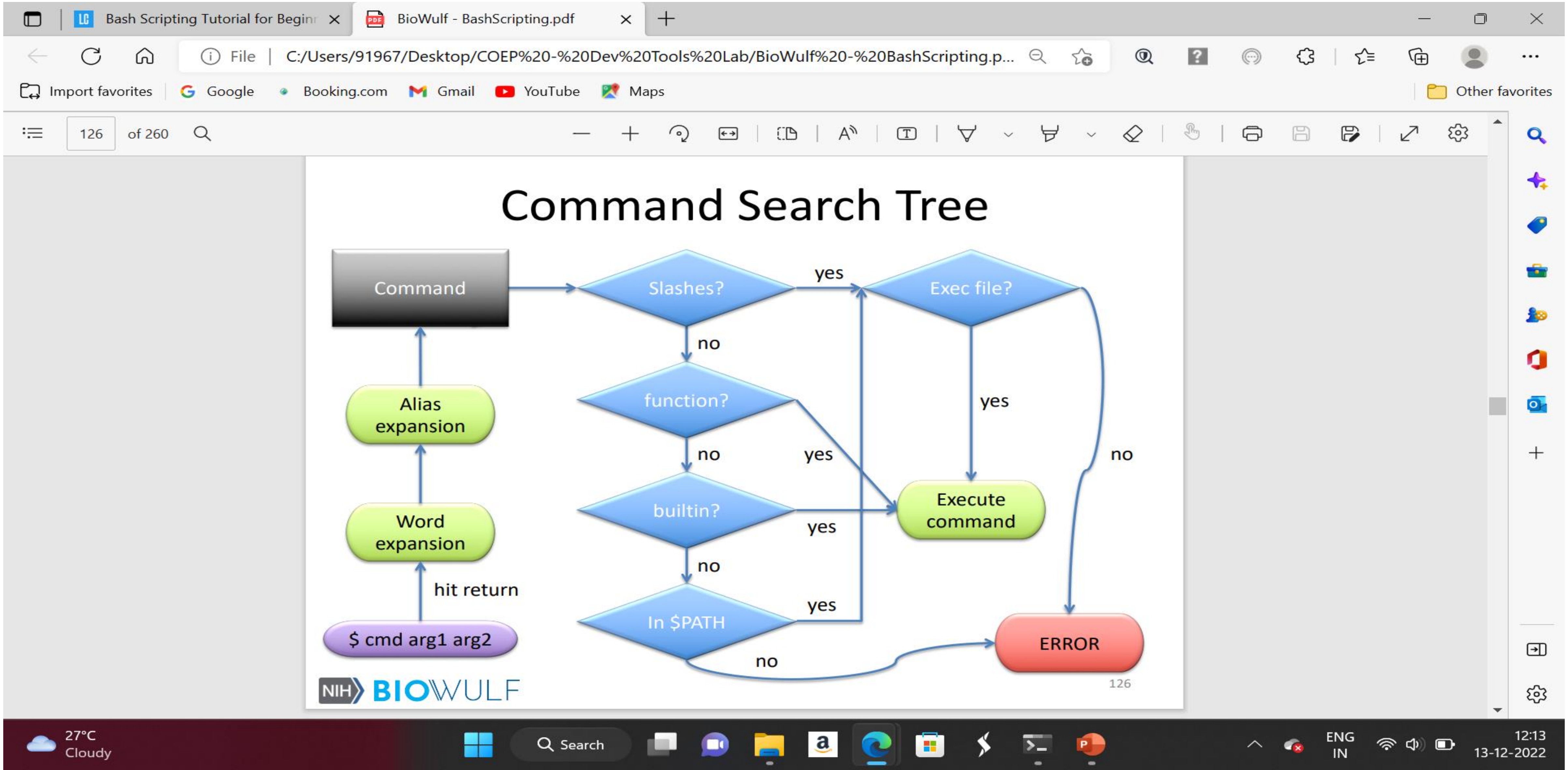
Scripting allows for an ***automatic commands execution*** that would otherwise be executed interactively one-by-one.

What is in a Script?

- Commands
- Variables
- Functions
- Loops
- Conditional Statements
- Comments and Documentation
- Options and Settings

SCRIPT EXECUTION

Command Execution



Script Execution – by calling bash

- create a script and inspect it’s contents :

```
$echo ‘echo Hello World! ’  > hello_world.sh
cat  hello_world.sh
```

Text (A small script) is redirected to a file

- call it with bash:

```
$bash hello_world.sh
```

Call bash – to interpret and execute the file

- results :

```
Hello World!
```

Multiline file creation

```
$ cat << ALLDONE >
hello_world_multiline.sh
> echo Hello World!
> echo Yet another line.
> echo This is getting boring.
> ALLDONE
$
```

```
$ bash hello_world_multiline.sh
Hello World!
Yet another line.
This is getting boring.
$
```

Before a command is executed, its input and output may be **redirected** using a special notation interpreted by the shell. Redirection may also be used to open and close files for the current shell execution environment.

Redirections are processed in the order they appear, from left to right. E.g. << first until ALLDONE, then > .

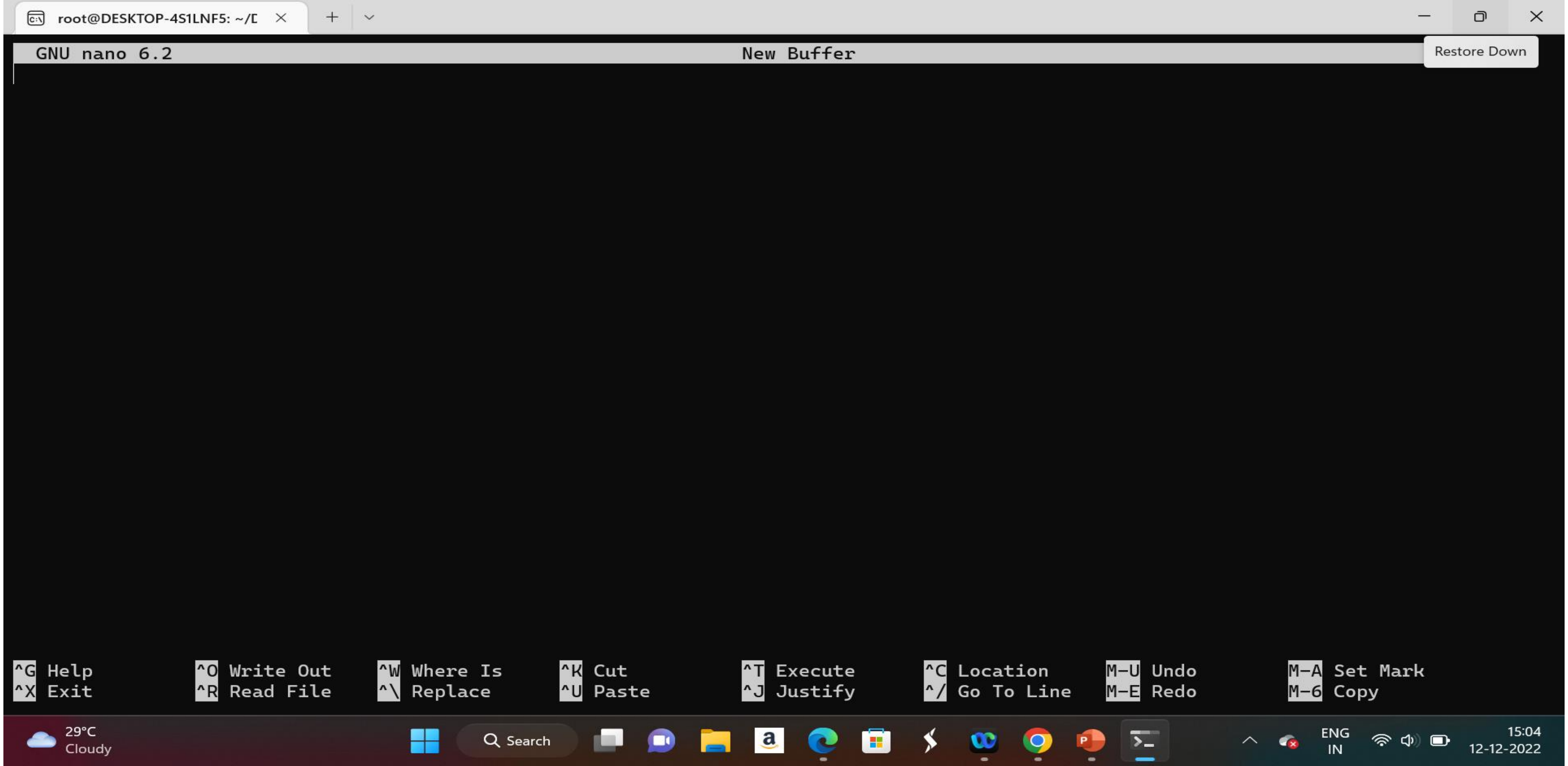
Here Documents (<<)

This type of redirection instructs the shell to read input from the current source until a line containing only delimiter (**with no trailing blanks**) is seen. **All of the lines read up to that point are then used as the standard input for a command**.

The format of here-documents is:

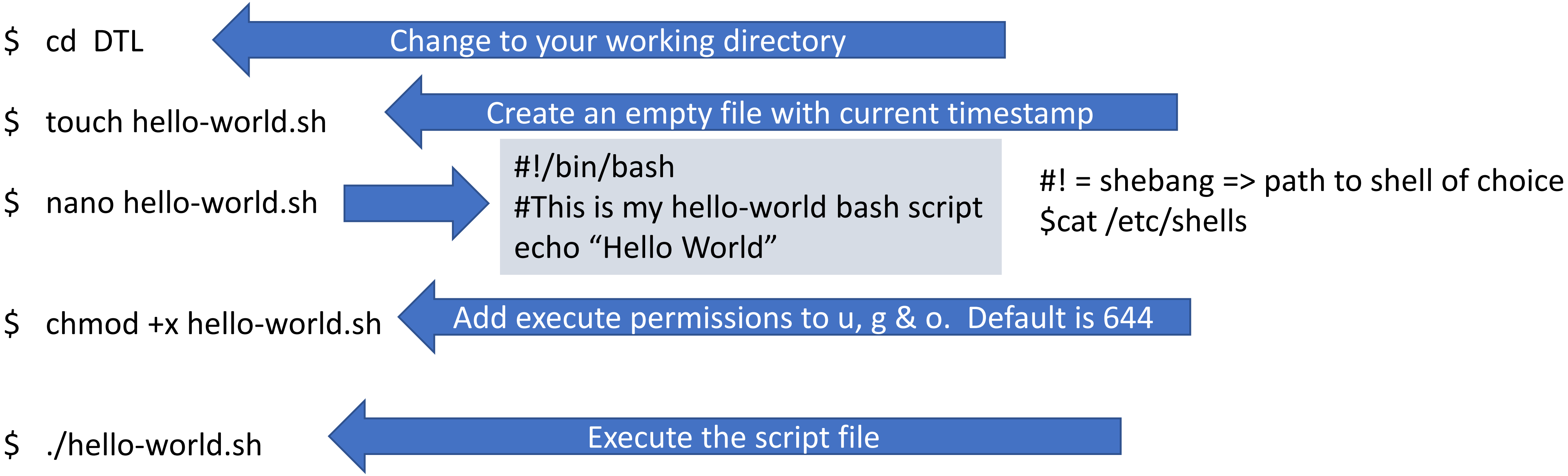
```
<<[-]word
        here-document
delimiter
```

nano ….!



```
  GNU nano 6.2                                    New Buffer





^G Help          ^O Write Out     ^W Where Is      ^K Cut           ^T Execute       ^C Location      M-U Undo         M-A Set Mark
^X Exit          ^R Read File     ^\ Replace       ^U Paste         ^J Justify       ^/ Go To Line    M-E Redo         M-6 Copy
```

Script Execution with #! - shebang



```
$   cd  DTL
```

Change to your working directory

```
$   touch hello-world.sh
```

Create an empty file with current timestamp

```
$   nano hello-world.sh
```

```
#!/bin/bash
#This is my hello-world bash script
echo “Hello World”
```

#! = shebang => path to shell of choice
$cat /etc/shells

```
$   chmod +x hello-world.sh
```

Add execute permissions to u, g & o. Default is 644

```
$   ./hello-world.sh
```

Execute the script file

Debugging

- Call bash with –x –v

```
$ bash –x –v hello_world.sh
```

```
#!/bin/bash –xv
echo Hello World!
```

- -x displays commands and their results
- -v displays everything, even comments and spaces
- -xv can be combined. In fact, for shebang they MUST be combined

Special Parameters

Positional Parameters

- Positional parameters are arguments passed to the shell when invoked
- Denoted by ${digit}, where digit > 0. $0 is the name of the script – e.g. x.sh

```
$ cat x.sh
 #!/bin/bash
echo ${4} ${15} ${7} ${3} ${1} ${20}
```

```
$ bash x.sh {A..Z}
 D O G C A T
```

EXPLANATION:

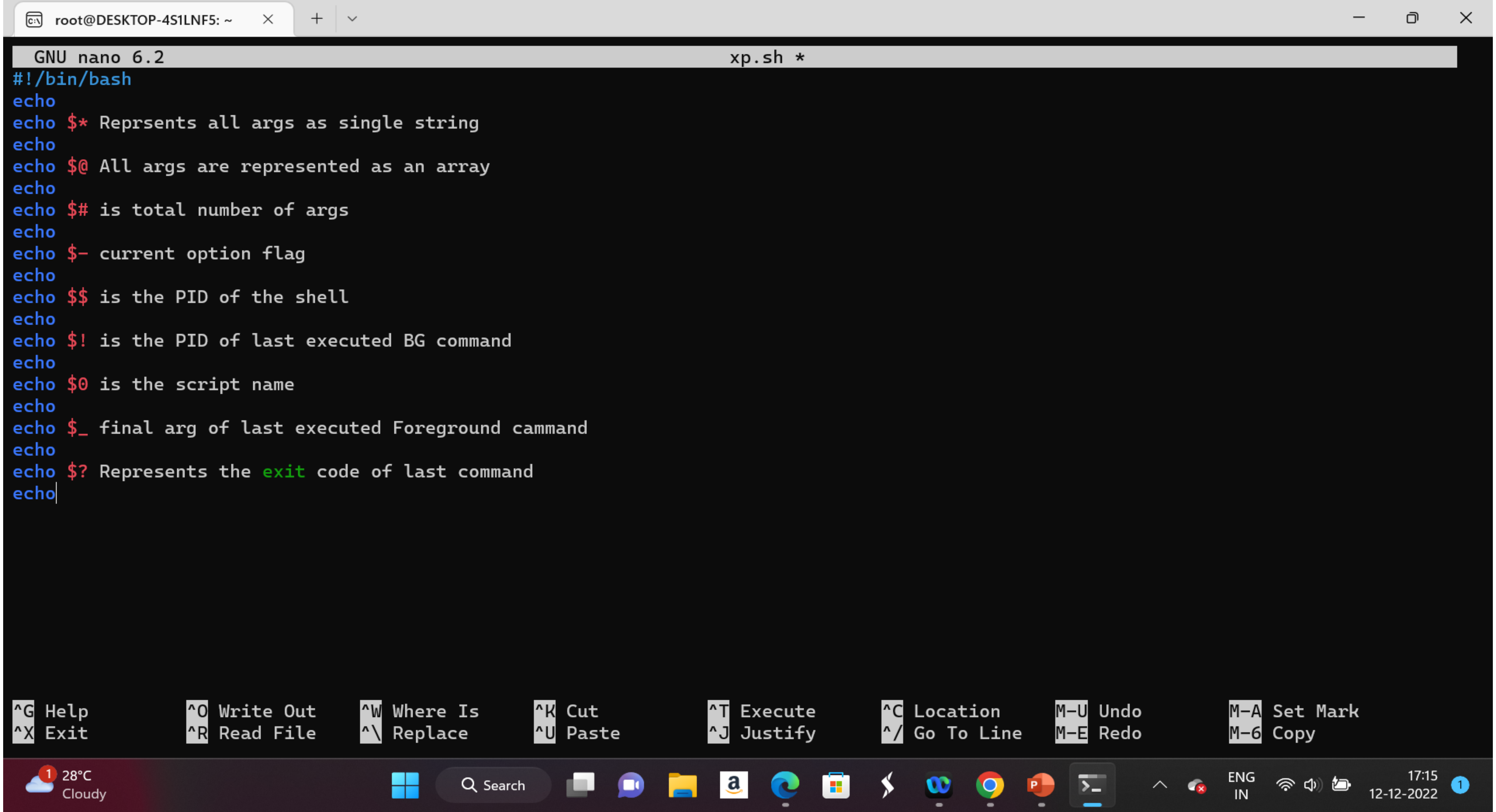
- {A..Z} is a case of Brace Expansion. Sequence Expansion
- The expanded seq is passed as an arg to x.sh
- ${digit} is selected “positionally” in the command, echo in this case.

Special Shell Parameters

Single character represents the parameter

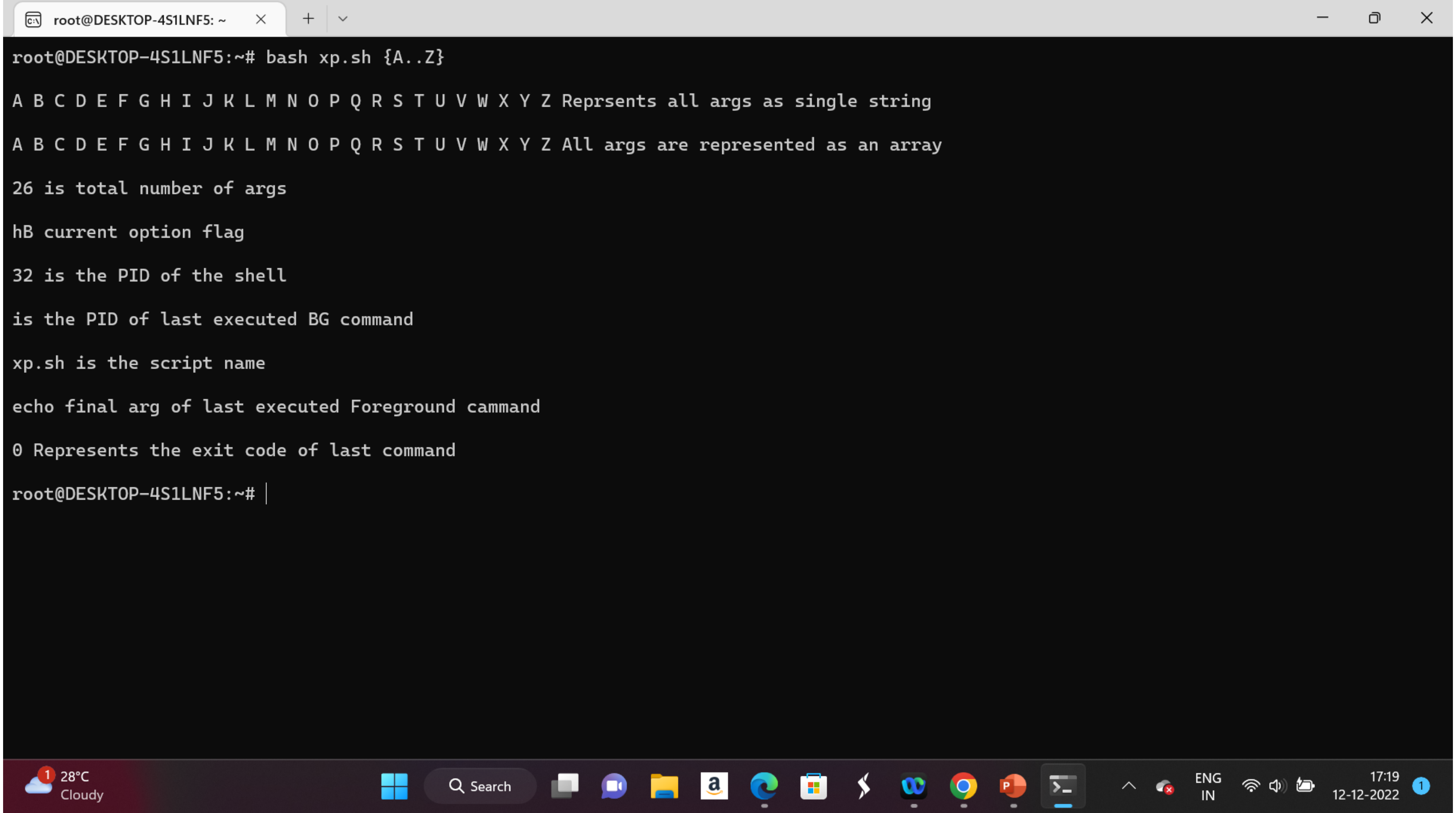
- $* - expands to all the positional parameters
- $@ - the same as $*, but as array rather than string
- $# - number of positional parameters
- $- - current option flags when shell invoked
- $$ - process id of the shell
- $! - process id of last executed background command
- $0 - name of the shell or shell script
- $_ - final argument of last executed foreground command
- $? - exit status of last executed foreground command

xp.sh – script to explore all special shell parameters



```
#!/bin/bash
echo
echo $* Reprsents all args as single string
echo
echo $@ All args are represented as an array
echo
echo $# is total number of args
echo
echo $- current option flag
echo
echo $$ is the PID of the shell
echo
echo $! is the PID of last executed BG command
echo
echo $0 is the script name
echo
echo $_ final arg of last executed Foreground cammand
echo
echo $? Represents the exit code of last command
echo
```

xp.sh – results



```
root@DESKTOP-4S1LNF5:~# bash xp.sh {A..Z}

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z Reprsents all args as single string

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z All args are represented as an array

26 is total number of args

hB current option flag

32 is the PID of the shell

is the PID of last executed BG command

xp.sh is the script name

echo final arg of last executed Foreground cammand

0 Represents the exit code of last command

root@DESKTOP-4S1LNF5:~# 
```

Shell Expansion

1 https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html

Brace Expansion:
Patterns to be brace expanded take the form of an optional *preamble*, followed by either a series of comma-separated strings or a sequence expression between a pair of braces, followed by an optional *postscript*. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right

EXAMPLES :

```
$echo a{b,d,c}e
abe ade ace

$echo {A..Z}
A B C ……….Z

mkdir z{1..10}
ls
```

Parameter Expansion
A parameter is an argument supplied to a command / function / script.

- $ is placed outside for parameter expansion

  EXAMPLE

  ```
  name=monster
  echo $name
  ```
- Braces can be used to preserve variable

  ```
  echo ${name}_silly
  ```
- More than one row

  ```
  $ var1=cookie
  $ var3=_is_silly
  $ echo
  ${var1}_${name}${var3}
  ```

Shell Expansion – Arithmetic Expansion

Arithmetic Expansion

- **INTEGERS ONLY**
- ((…)) => is used to evaluate math
- **$ is a must**- for parameter expansion

    $ echo ((12 – 7)) is wrong
    $ echo $((12 – 7)) is correct

Variables can be updated, not just evaluated

```
a=4
b=8
echo $((a=a+b)
echo $a
```

The ++ and -- operators only work on variables, and update the value

```
a=4
((a++))
echo $a

b=8
unset b
((b--))
echo $b
```

Let and ((…)) are equivalent

```
a=1
let a++
echo $a
```

Shell Expansion – Command Substitution

- Command Substitution => substitute the value returned by `command …` or $(command …)
- Better to use $(…) than `…`

Try out 3 different echo

```
echo uname –n

echo `uname –n`

echo $(uname –n)
```

File names and permissions

Permissions.

- System default permissions are 777 for directories and 666 for files. Usually, umask is 0022. So, usually files have 644 permissions, when they are created, i.e. –rw-r- - r - - .
- To make shell scripts executable => chmod +x <script_file>

File extensions have no meaning to the system. They are mainly for your own reasons.

To see what is your file type:

```
$file <file_name>
```

VARIABLES

Variables

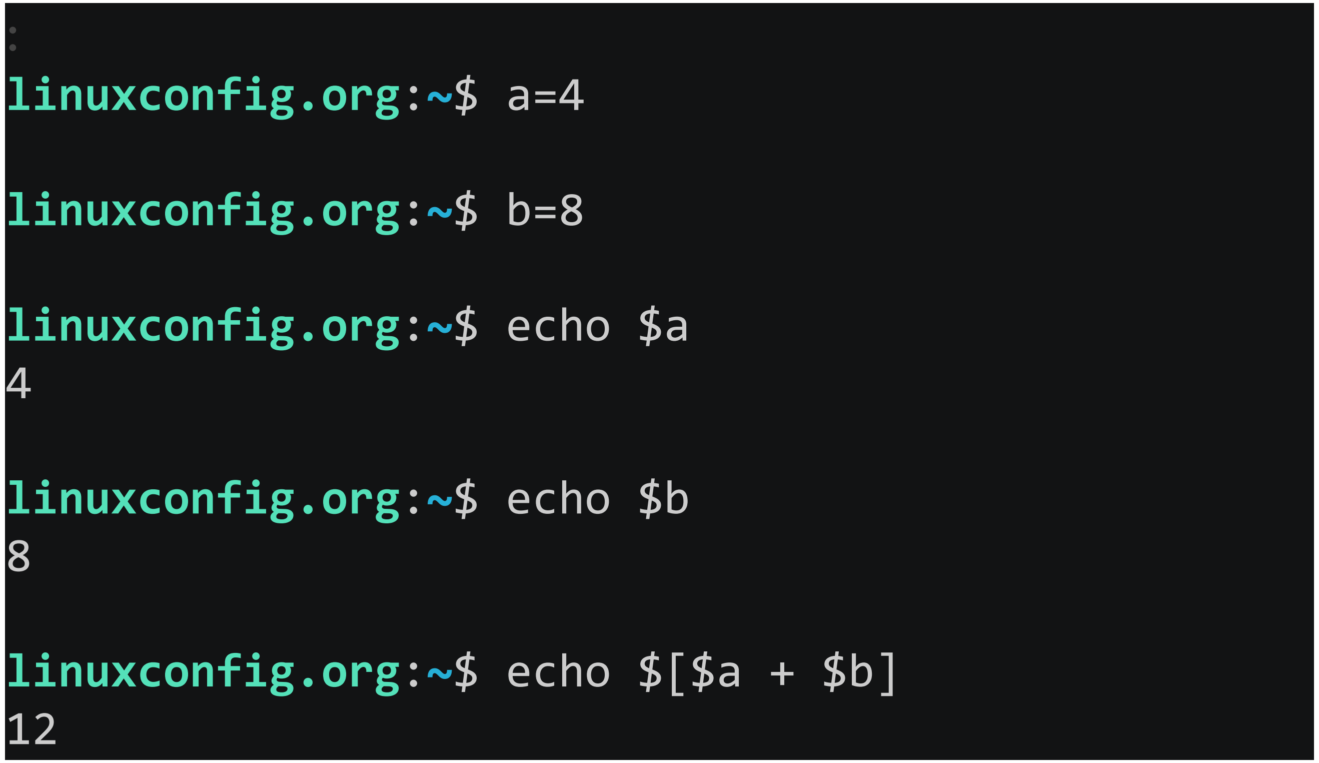
Variables => names for ‘values’ stored at specific memory locations. The programmer can store data in those locations, alter them and reuse them throughout the script

```
#!/bin/bash

greeting="Welcome"
user=$(whoami)
day=$(date +%A)

echo "$greeting back $user! Today is $day, which is the best day of the week!"
echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"
```

NOTES:

- $variable_name => access the value currently assigned to variable_name
- greeting is a variable with (constant) string assigned to it
- user and day are “COMMAND SUBSTITUTION” variables. The value assigned to them is generated by a the command in brackets.
- $BASH_VERSION is an internal / **shell** variable. Such variables are in CAPITALS, to distinguish from user defined variables

Variables can be defined in the command line



```
linuxconfig.org:~$ a=4

linuxconfig.org:~$ b=8

linuxconfig.org:~$ echo $a
4

linuxconfig.org:~$ echo $b
8

linuxconfig.org:~$ echo $[$a + $b]
12
```

Variables can also be used directly on the terminal's command line.

- First variables **a** and **b** are declared with integer data.
- Using **echo** command, we can print their values or even perform an arithmetic operation, Note **echo is required to see the values**. A command is necessary !
- **The $ is a must to interpret as a variable.**
- [EXPRESSION] is = test EXPRESSION
- $[EXPRESSION] = value returned by EXPRESSION?

EXIT CODES $?

backup.sh - A script to backup any user’s home directory

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

tar -czf $output $input

echo "Backup of $input completed! Details about the output backup
file:“
ls -l $output
```

Two new bash scripting concepts are introduced:

- Firstly, our new **backup.sh** script contains comment line. Every line starting with **#** sign except shebang will not be interpreted by bash and will only serve as a programmer’s internal note.
- Secondly, the script uses a new shell scripting trick **${parameter}** called parameter expansion. In our case, curly braces **{}** are required because our variable **$user** *is preceded by prefix followed by a suffix*. The suffix includes $(date) – a command substitution.

The script will no longer be bind to a specific user. From now on our **backup.sh** bash script can be run by any user (others have r-x permission). while still backing up a correct user home directory

Three types of variables, based on scope

Local variables : Scope only within the instance of the script

Environment variables (ENVs) : Can have global / local scope.

To see ENV variables $env
e.g. PATH => all executables located

Global ENVs => accessible by all any process in the environment of the terminal.
$cat /etc/environment
$cat /etc/profile
$cat /etc/bash.bashrc

Local ENVs => specific scripts of the user
e.g. ~/.bashrc , ~/.bash_profile, ~/.bash_login, ~/.profile

Shell variables : variables required for proper functioning of all shell scripts. They are a mix of Global and Local ENVs

Setting the environment

- The environment is a set of variables and functions recognized by the kernel and used by most programs
- Not all variables are environment variables, must be exported
- Initially set by startup files
- **printenv** displays variables and values in the environment
- **set** - Set or unset values of shell options and positional parameters.

Arrays – special type of variables

- An array is a linear, ordered set of values.
- The values are indexed by integers
- Arrays are referenced by {} **and** []

```
array=(a b c)          => note ( seq w/o ,), to declare array

echo ${array[*]}       => a b c . $ for variable. {} for braces expansion. [] for indexing

echo ${array[2]}       => 0 start indexing. So, c .

echo ${#array[*]}      => NOTE #
```

```
# Loop thru an array - NOTE [@] => looping thru all elements

for chr in ${array[@]}; do
 echo $chr
done
```

[*} vs [@}

- [*] => concatenates all elements

```
for chr in “${array[@]}”; do  echo $chr ; done
 vs
for chr in “${array[*]}”; do  echo $chr ; done
```

INPUT AND OUTPUT

Redirection

- Every process has three file descriptors (file handles): STDIN (0), STDOUT (1), STDERR (2)
- Content can be redirected

`cmd < x.in`

=> Redirect file descriptor 0 to x.in

`cmd > x.out`

=> Redirect file descriptor 1 to x.out

`cmd 1> x.out 2> x.err`

⇒ Redirect file descriptor 1 from STDOUT to x.out,
file descriptor 2 from STDERR to x.err

`cmd > x.out 2>&1`

⇒ Redirect file descriptor 1 from STDOUT to x.out,
then, redirect file descriptor 2 from STDERR to wherever descriptor 1 is pointing, i.e. x.out

`cmd >> x.out`

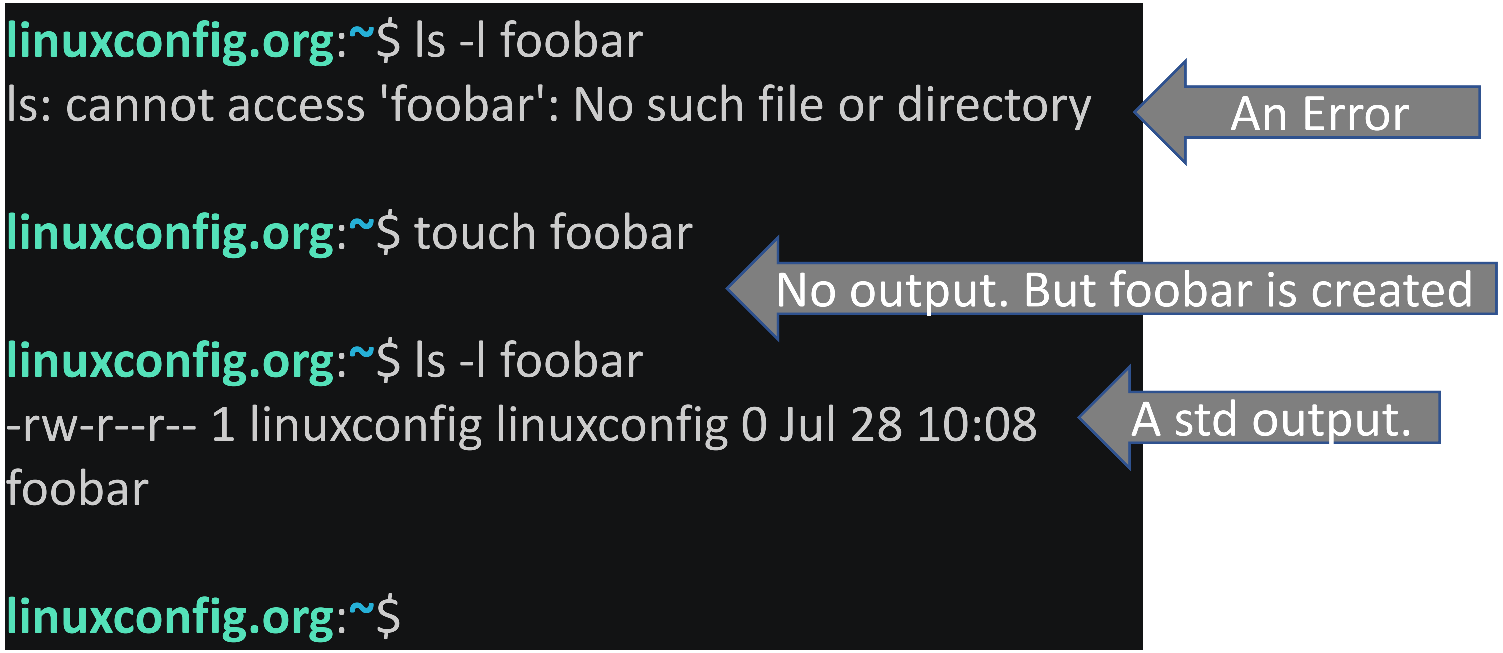
=> Append STDOUT to x.out

`cmd 1>> x.out 2>&1`

⇒ Combine STDOUT and STDERR. Append to x.out

ORDERING of redirections matter -> i.e. it is left to right

Input, Output and Error Redirections



```
linuxconfig.org:~$ ls -l foobar
ls: cannot access 'foobar': No such file or directory

linuxconfig.org:~$ touch foobar

linuxconfig.org:~$ ls -l foobar
-rw-r--r-- 1 linuxconfig linuxconfig 0 Jul 28 10:08
foobar

linuxconfig.org:~$
```

An Error

No output. But foobar is created

A std output.

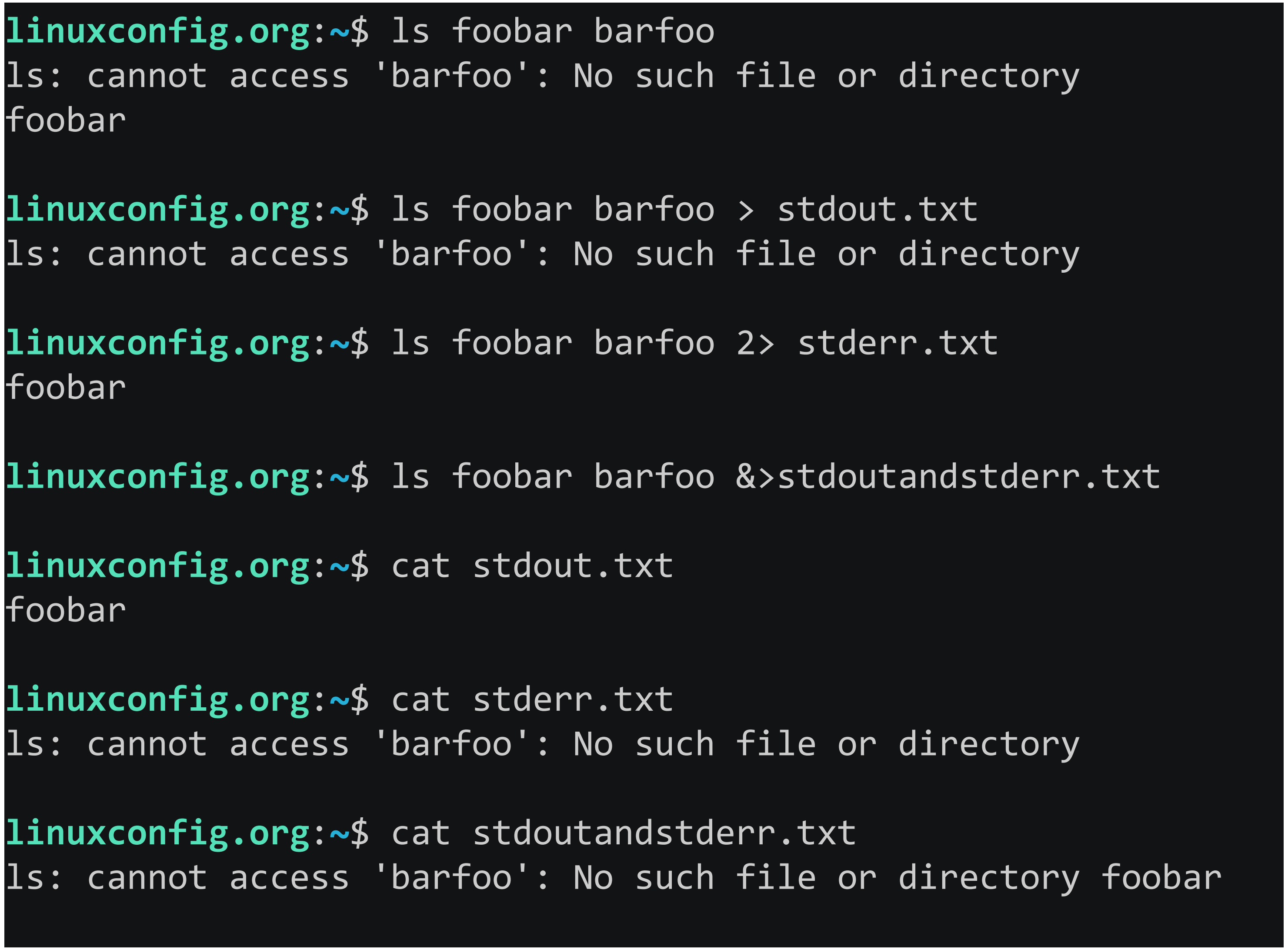
A NOTE ON EXIT STATUS

- echo $?
    - 0 => success
    - Non-zero => failure
        - 127 => command not found
        - 126 => command found, but not executable

Redirecting stdout and stderr

The difference between **stdout** and **stderr** output is an essential concept. It allows us to **redirect each output separately**.

- The **>** notation is used to redirect **stdout** to a file (say stdout.txt), instead of stdout (i.e TTY)
- The **2>** notation is used to redirect **stderr,** and not TTY.
- And **&>** is used to redirect both **stdout** and **stderr**. No TTY output.
- The **cat** command is used to display a content of any given file



```
linuxconfig.org:~$ ls foobar barfoo
ls: cannot access 'barfoo': No such file or directory
foobar

linuxconfig.org:~$ ls foobar barfoo > stdout.txt
ls: cannot access 'barfoo': No such file or directory

linuxconfig.org:~$ ls foobar barfoo 2> stderr.txt
foobar

linuxconfig.org:~$ ls foobar barfoo &>stdoutandstderr.txt

linuxconfig.org:~$ cat stdout.txt
foobar

linuxconfig.org:~$ cat stderr.txt
ls: cannot access 'barfoo': No such file or directory

linuxconfig.org:~$ cat stdoutandstderr.txt
ls: cannot access 'barfoo': No such file or directory foobar
```

The backup.sh script with error redirection

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

tar -czf $output $input 2> /dev/null

echo "Backup of $input completed! Details about the output backup
file:"
ls -l $output
```

We can eliminate this unwanted **stderr** message by redirecting it with **2>** notation to **/dev/null**.

Imagine **/dev/null** as a data sink, which discards any data redirected to it.

Data written to /dev/null is discarded (like Trash)

Redirecting stdin

Normally, terminal input comes from a keyboard. Any keystroke you type is accepted as **stdin**.
The alternative method is to accept command input from a file using **<** notation.

Consider the following example where we first feed cat command from the keyboard and redirecting the output to **file1.txt**. Later, we allow cat command to read the input from **file1.txt** using **<** notation

```
linuxconfig.org:~$ cat > file1.txt
I am using keyboard to input text.
Cat command reads my keyboard input, converts it to stdout which is instantly redirected to file1.txt
That is, until I press CTRL+D

linuxconfig.org:~$ cat < file1.txt
I am using keyboard to input text.
Cat command reads my keyboard input, converts it to stdout which is instantly redirected to file1.txt
That is, until I press CTRL+D
```

FUNCTIONS

Functions

Functions are a set of commands, grouped as a single function. This can be extremely useful if the output or calculation you require consists of multiple commands, and it will be expected multiple times throughout the script execution. Functions are defined by using the function keyword and followed by function body enclosed by curly brackets

```
$function status { date; uptime; who | grep $USER; }
$status
```

```
$declare –f status  => displays code for the function
```

```
$export –f status  => the function can be propagated to child shells
```

```
$unset status  => the function is deleted
```

A code example – using function

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.
user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

# The function total_files reports number of files for a directory.
function total_files { find $1 -type f | wc –l }

# The function total_directories reports a total number of directories for a
# given directory.
function total_directories {  find $1 -type d | wc –l  }

tar -czf $output $input 2> /dev/null
```

```
echo -n "Files to be included:"
total_files $input
echo -n "Directories to be included:"
total_directories $input

echo "Backup of $input completed!"

echo "Details about the output backup file:"
ls -l $output
```

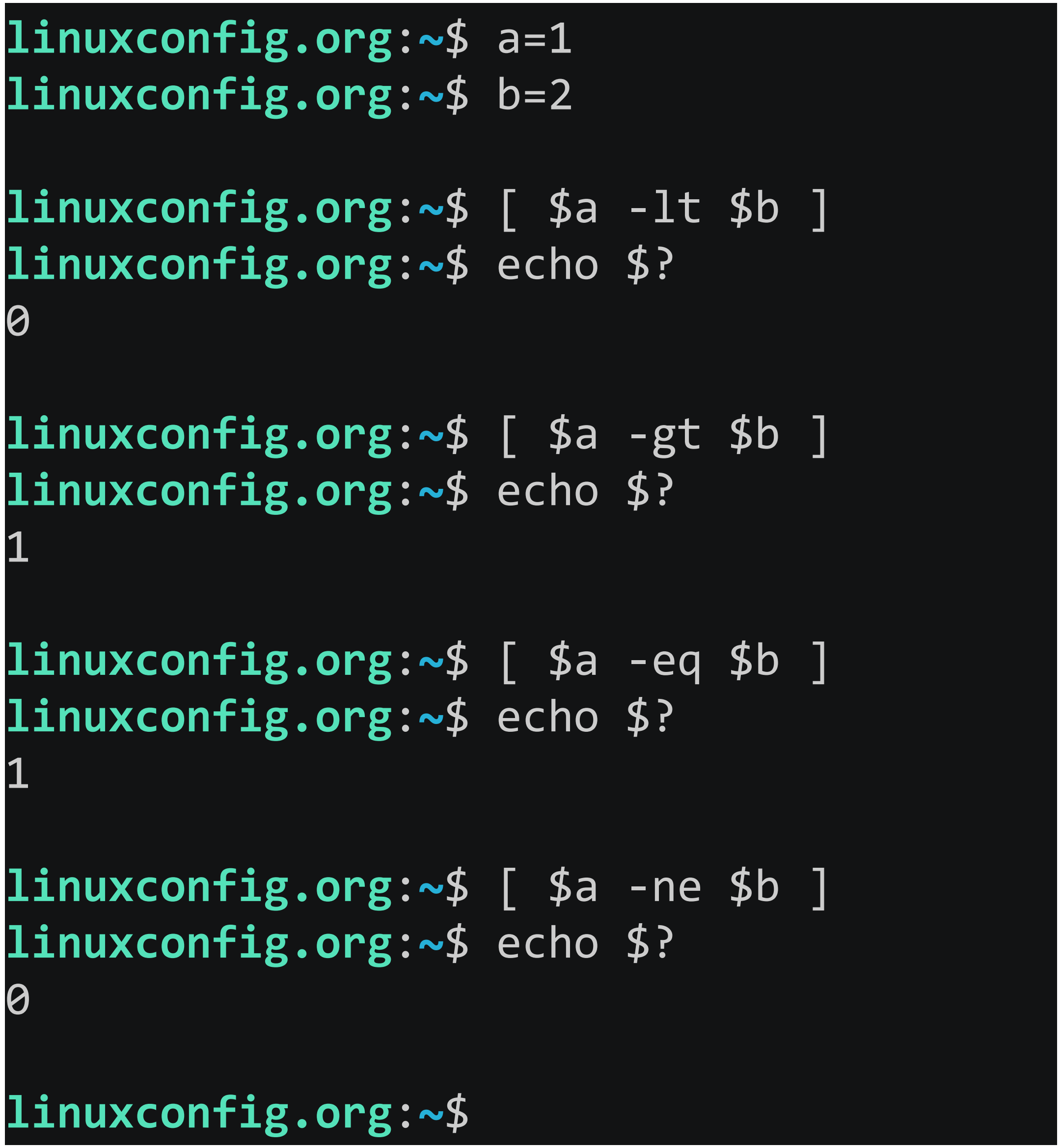
NUMERIC AND STRING COMPARISONS

Numeric and String Comparisons

Using comparisons, we can compare strings (words, sentences) or integer numbers whether raw or as variables. The following table lists rudimentary comparison operators for both numbers and strings:

Description	Numeric Comparison	String Comparison
less than	-lt	<
greater than	-gt	>
equal	-eq	=
not equal	-ne	!=
less or equal	-le	N/A
greater or equal	-ge	N/A

Numeric Comparison



```
linuxconfig.org:~$ a=1
linuxconfig.org:~$ b=2

linuxconfig.org:~$ [ $a -lt $b ]
linuxconfig.org:~$ echo $?
0

linuxconfig.org:~$ [ $a -gt $b ]
linuxconfig.org:~$ echo $?
1

linuxconfig.org:~$ [ $a -eq $b ]
linuxconfig.org:~$ echo $?
1

linuxconfig.org:~$ [ $a -ne $b ]
linuxconfig.org:~$ echo $?
0

linuxconfig.org:~$
```

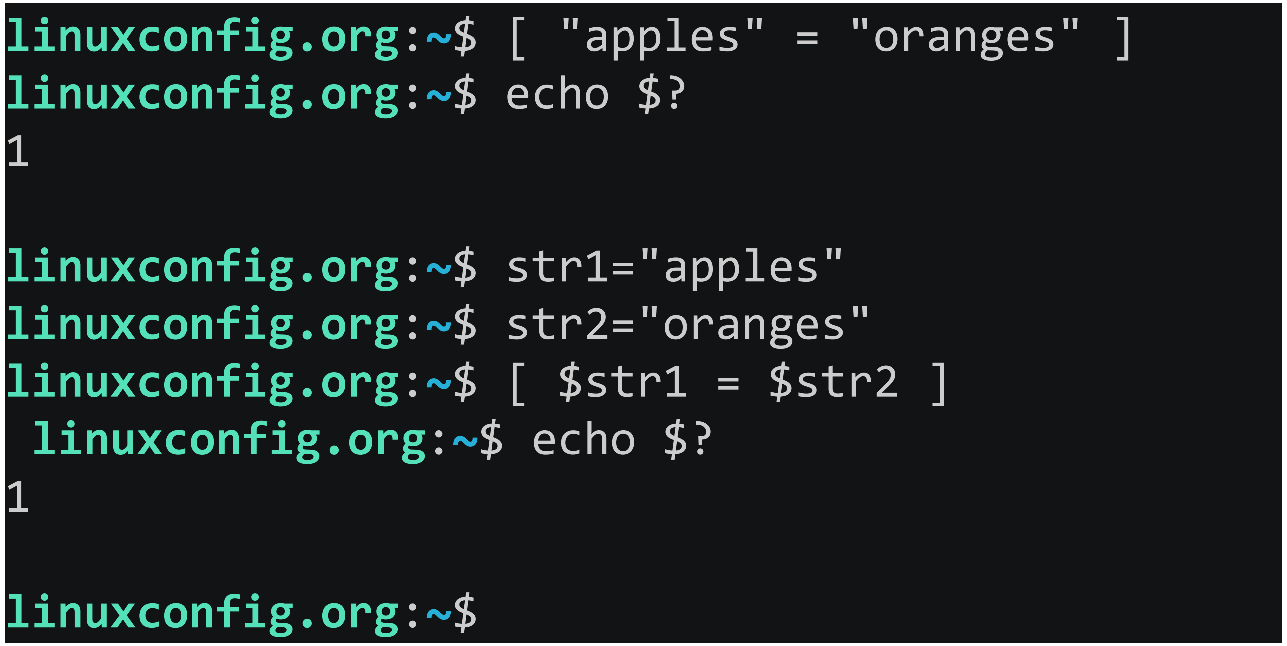
Say, we would like to compare numeric values like two integers **1** and **2**. The alongside example will first define two variables **$a** and **$b** to hold our integer values.

Next, we use square brackets and numeric comparison operators to perform the actual evaluation. NOTE : [EXPR] is –eq to test if T/F .

Using **echo $?** command, we check for a return value of the previously executed evaluation. There or two possible outcomes for every evaluation, true or false. If the return value is equal to **0**, then the comparison evaluation is true. However, if the return value is equal to **1**, the evaluation resulted as false.

NOTE : echo $? Must be used. Else you will get error status 127

Comparing strings



```
linuxconfig.org:~$ [ "apples" = "oranges" ]
linuxconfig.org:~$ echo $?
1

linuxconfig.org:~$ str1="apples"
linuxconfig.org:~$ str2="oranges"
linuxconfig.org:~$ [ $str1 = $str2 ]
 linuxconfig.org:~$ echo $?
1

linuxconfig.org:~$
```

Code example using comparisons

```
#!/bin/bash

string_a="UNIX“
 string_b="GNU"
echo "Are $string_a and $string_b strings equal?" [ $string_a = $string_b ]
echo $?

num_a=100
num_b=100
echo "Is $num_a equal to $num_b ?" [ $num_a -eq $num_b ]
echo $?
```

CONDITIONAL STATEMENTS

If .. elif .. else .. fi

```
if test-commands ; then
    consequent-commands
elif more-test-commands ; then
    more-consequents
else
    alternate-consequents
fi
```

```
# EXAMPLE CODE
if test -e ~/.bashrc; then
        echo "~/.bashrc exists"
elif test -e ~/.bash_profile; then
        echo "~/.bash_profile exists"
else echo "You may not be running bash"
fi
```

[and [[are substitute for the test command

[is an executable file

```
which [ => /usr/bin/[
```

Backup script again- with conditionals

```
#!/bin/bash

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

function total_files { find $1 -type f | wc –l  }

function total_directories { find $1 -type d | wc –l  }

function total_archived_directories {
	tar -tzf $1 | grep  /$ | wc -l
}

function total_archived_files { tar -tzf $1 | grep -v /$ | wc –l  }

tar -czf $output $input 2> /dev/null
```

```
src_files=$( total_files $input )
src_directories=$( total_directories $input )

arch_files=$( total_archived_files $output )
arch_directories=$( total_archived_directories $output )

echo "Files to be included: $src_files"
echo "Directories to be included: $src_directories"
echo "Files archived: $arch_files"
echo "Directories archived: $arch_directories"

if [ $src_files -eq $arch_files ]; then
	echo "Backup of $input completed!"
	echo "Details about the output backup file:"
	ls -l $output
else
	echo "Backup of $input failed!"
fi

```

Blank Line after fi required

Conditional for Arithmetic expressions

```
a=4
if ((a==4)) ; then echo yes ; else echo no ; fi
```

yes

```
if (( (a-5) == 0 )) ; then echo yes ; else echo no ; fi
```

no

```
if (( a < 10  )) ; then echo yes ; else echo no ; fi
```

yes

BE CAREFUL : == and = are not the same

```
a=4
if (( a = 5  )) ; then echo yes ; else echo no ; fi
```

yes

Bash Loops – Looping Constructs

GENERAL NOTE : wherever a ‘;’ appears in the description of a command’s syntax, it may be replaced with one or more newlines

until

Syntax : until *test-commands*; do *consequent-commands*; done

Execute *consequent-commands* as long as *test-commands* has an exit status which is not zero. The return status is the exit status of the last command executed in *consequent-commands*, or zero if none was executed.

while

Syntax : while *test-commands*; do *consequent-commands*; done

Execute *consequent-commands* as long as *test-commands* has an exit status of zero. The return status is the exit status of the last command executed in *consequent-commands*, or zero if none was executed.

for

Syntax : for *name* [[in [*words* …]] ;] do *commands*; done

Expand *words* (see Shell Expansions), and execute *commands* once for each member in the resultant list

big_backup.sh

```
#!/bin/bash

# This bash script is used to backup a user's home
directory to /tmp/.

function backup {

  if [ -z $1 ]; then
        user=$(whoami)
  else
        if [ ! -d "/home/$1" ]; then
                echo "Requested $1 user home
directory doesn't exist."
                exit 1
        fi
        user=$1
  fi
```

[-z $1]

- -z checks 1st **positional** parameter supplied to backup function zero length. -z returns TRUE if zero

[!-d “/home/$1”]

- By default –d returns true if directory exists. So, !-d => no directory exists.
- If [No dir] exit with error status 1

big_backup.sh …contd

```
 input=/home/$user
   output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

   function total_files {
             find $1 -type f | wc -l
   }

   function total_directories {
             find $1 -type d | wc -l
   }

   function total_archived_directories {
             tar -tzf $1 | grep  /$ | wc -l
   }

   function total_archived_files {
             tar -tzf $1 | grep -v /$ | wc -l
   }
```

```
    tar -czf $output $input 2> /dev/null

   src_files=$( total_files $input )
   src_directories=$( total_directories $input )

   arch_files=$( total_archived_files $output )
   arch_directories=$( total_archived_directories $output )

   echo "########## $user ##########"
   echo "Files to be included: $src_files"
   echo "Directories to be included: $src_directories"
   echo "Files archived: $arch_files"
   echo "Directories archived: $arch_directories"
```

Big_backup.sh … contd

```
 if [ $src_files -eq $arch_files ]; then
             echo "Backup of $input completed!"
             echo "Details about the output backup file:"
             ls -l $output
   else
             echo "Backup of $input failed!"
   fi
}

for directory in $*; do
   backup $directory
done;
```

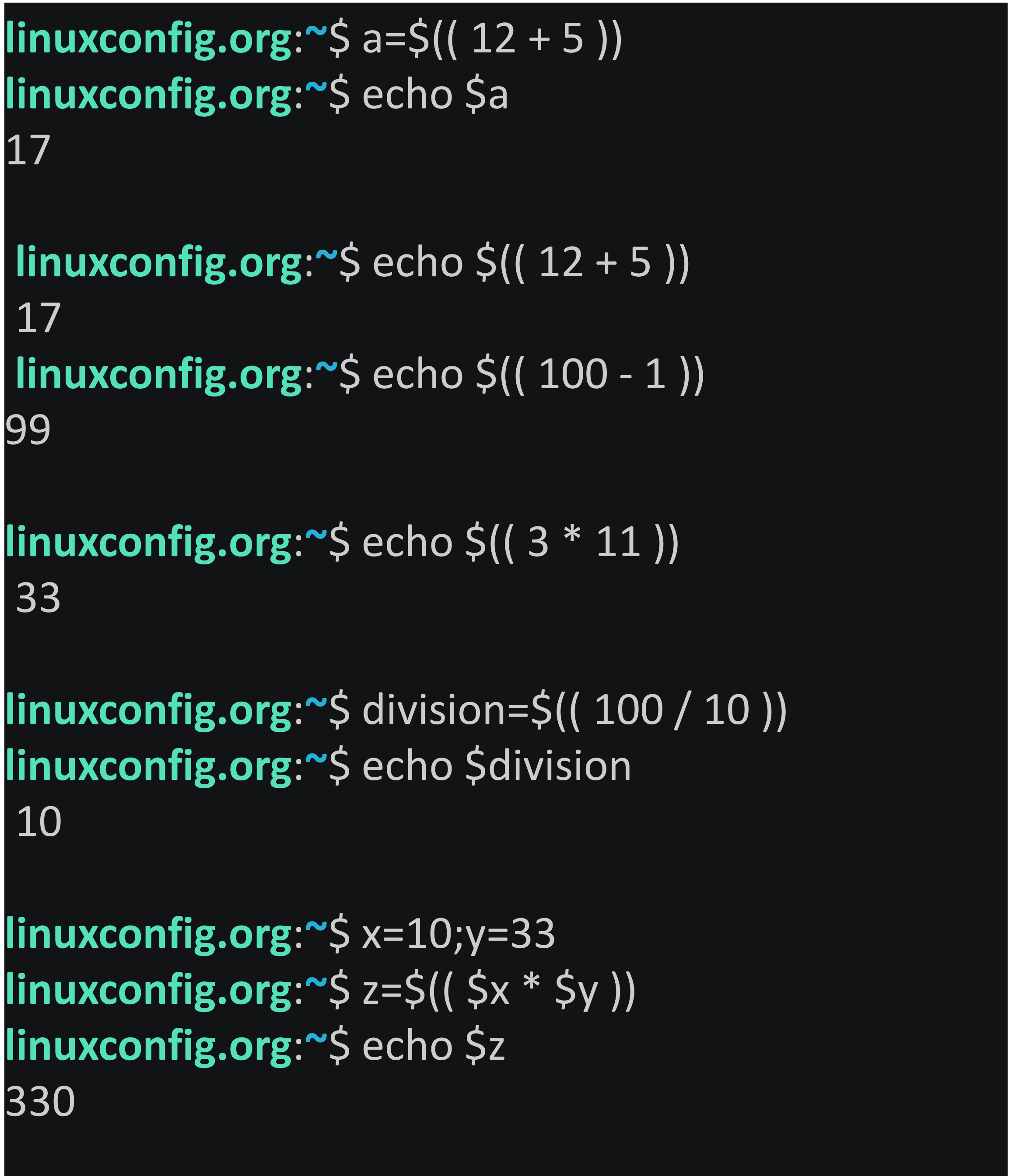
To execute, you need to provide $1, which is typically a $user, e.g. :

./big_backup.sh ramesh Sheela jenny

Note **for loop =>** for multiple users, create the backup of their home directory(s)**.** **$*** parameter expansion

BASH ARITHMETICS

Arithmetic Expansion



```
linuxconfig.org:~$ a=$(( 12 + 5 ))
linuxconfig.org:~$ echo $a
17

 linuxconfig.org:~$ echo $(( 12 + 5 ))
 17
 linuxconfig.org:~$ echo $(( 100 - 1 ))
99

linuxconfig.org:~$ echo $(( 3 * 11 ))
 33

linuxconfig.org:~$ division=$(( 100 / 10 ))
linuxconfig.org:~$ echo $division
 10

linuxconfig.org:~$ x=10;y=33
linuxconfig.org:~$ z=$(( $x * $y ))
linuxconfig.org:~$ echo $z
330
```

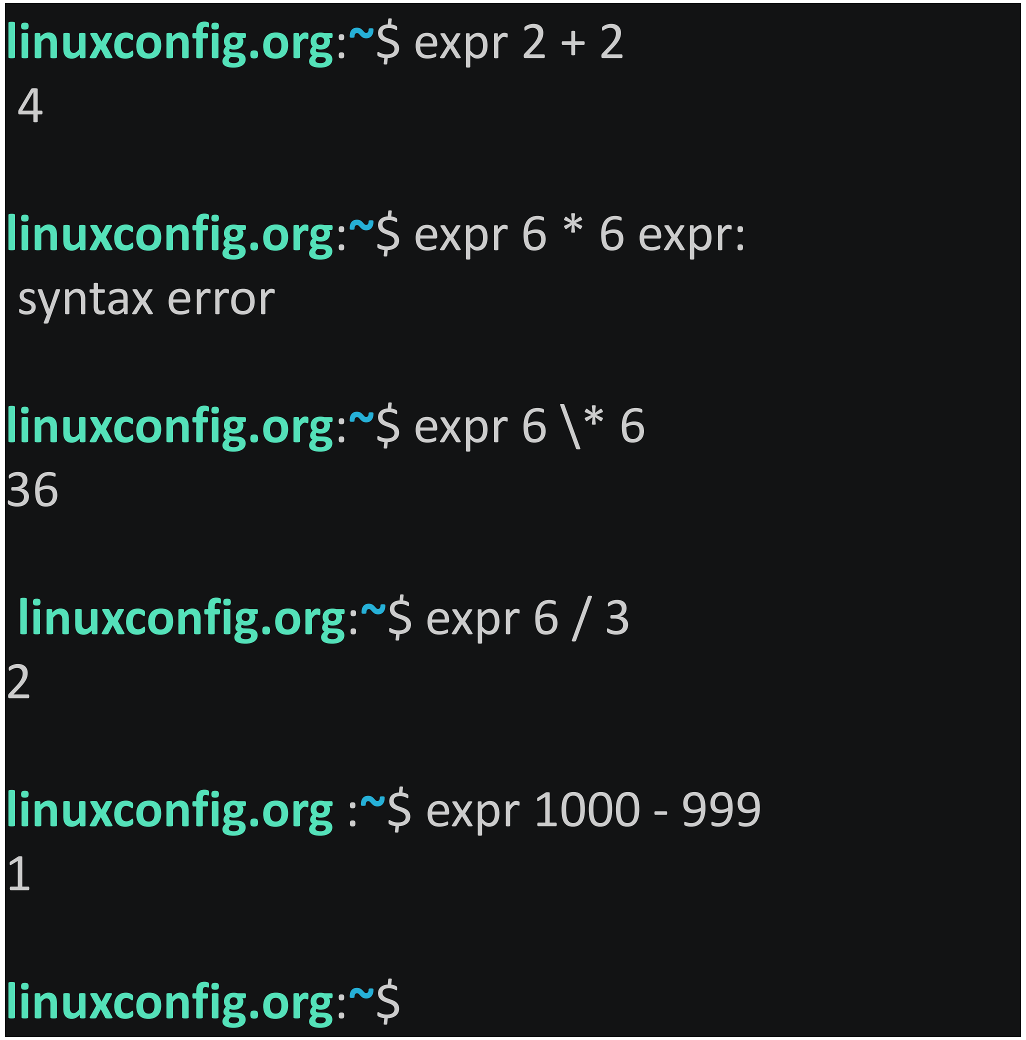
The arithmetic expansion is the simplest method to achieve basic calculations.

We just enclose any mathematical expression inside **double parentheses**.

Alongside => simple addition, subtraction, multiplication and division calculations with **integers**.

> To do floating point arithmetic utilies such as **bc** is required.

expr command



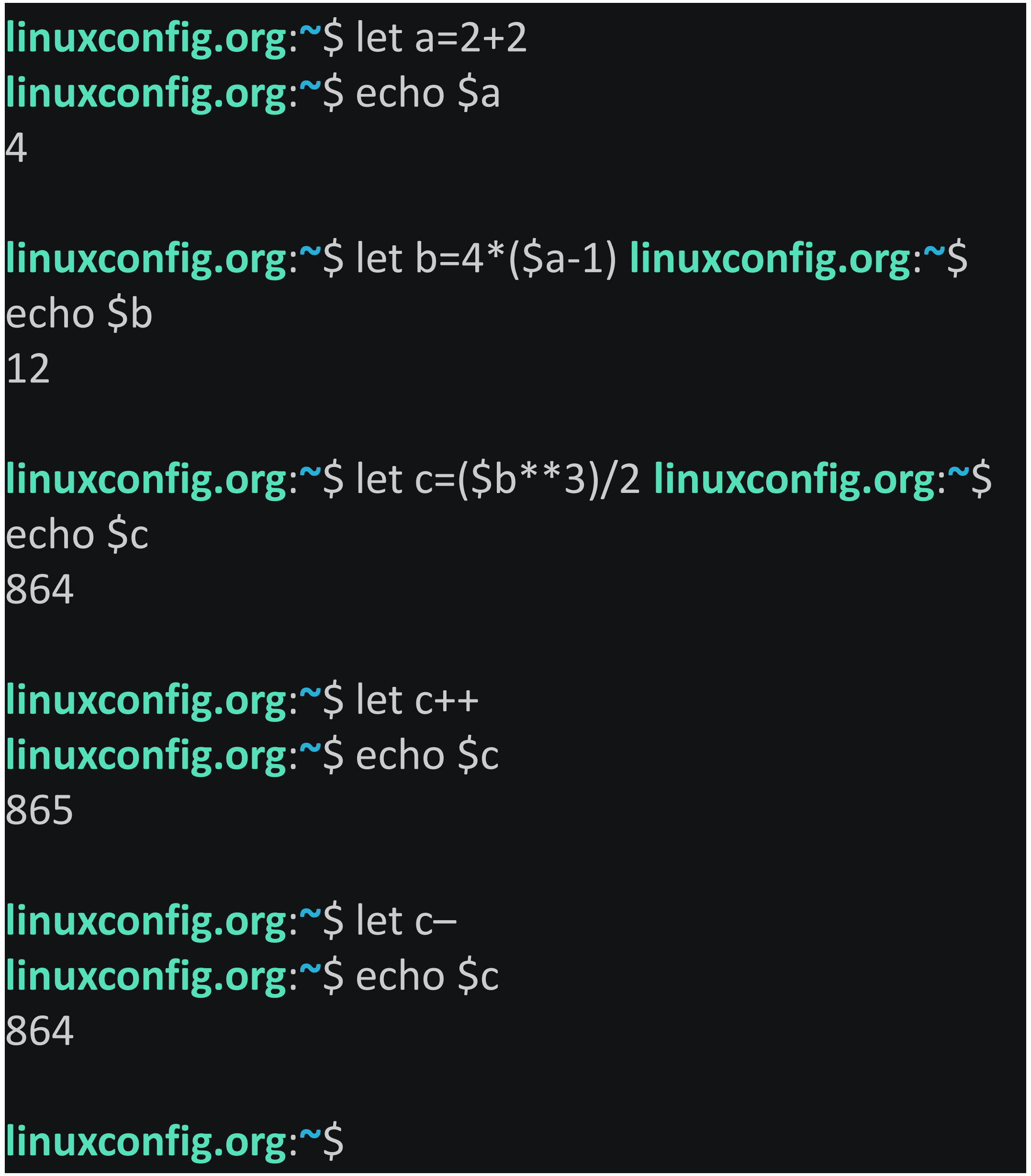
```
linuxconfig.org:~$ expr 2 + 2
 4

linuxconfig.org:~$ expr 6 * 6 expr:
 syntax error

linuxconfig.org:~$ expr 6 \* 6
36

 linuxconfig.org:~$ expr 6 / 3
2

linuxconfig.org :~$ expr 1000 - 999
1

linuxconfig.org:~$
```

Using the expr command allows us to perform an arithmetic operation even without enclosing our mathematical expression within brackets.

NOTE:

- Do not forget to escape asterisk multiplication sign to avoid **expr: syntax error**
- Divide is ‘/’

let command



```
linuxconfig.org:~$ let a=2+2
linuxconfig.org:~$ echo $a
4

linuxconfig.org:~$ let b=4*($a-1) linuxconfig.org:~$
echo $b
12

linuxconfig.org:~$ let c=($b**3)/2 linuxconfig.org:~$
echo $c
864

linuxconfig.org:~$ let c++
linuxconfig.org:~$ echo $c
865

linuxconfig.org:~$ let c–
linuxconfig.org:~$ echo $c
864

linuxconfig.org:~$
```

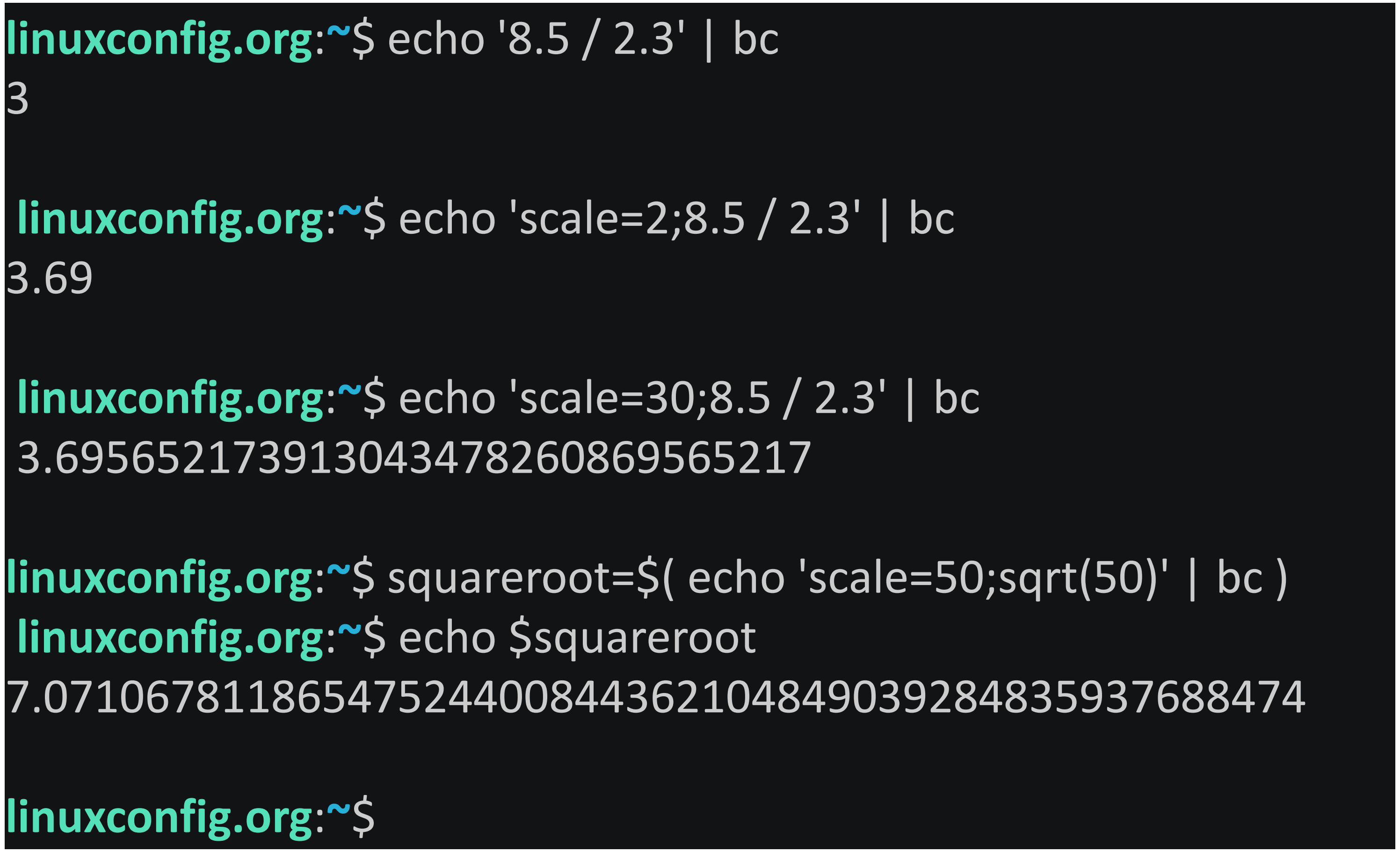
Similar to **expr** command, we can perform bash arithmetic operations with **let** command.

let command evaluates a mathematical expression and stores its result into a variable.

See alongside examples of **let** **:**

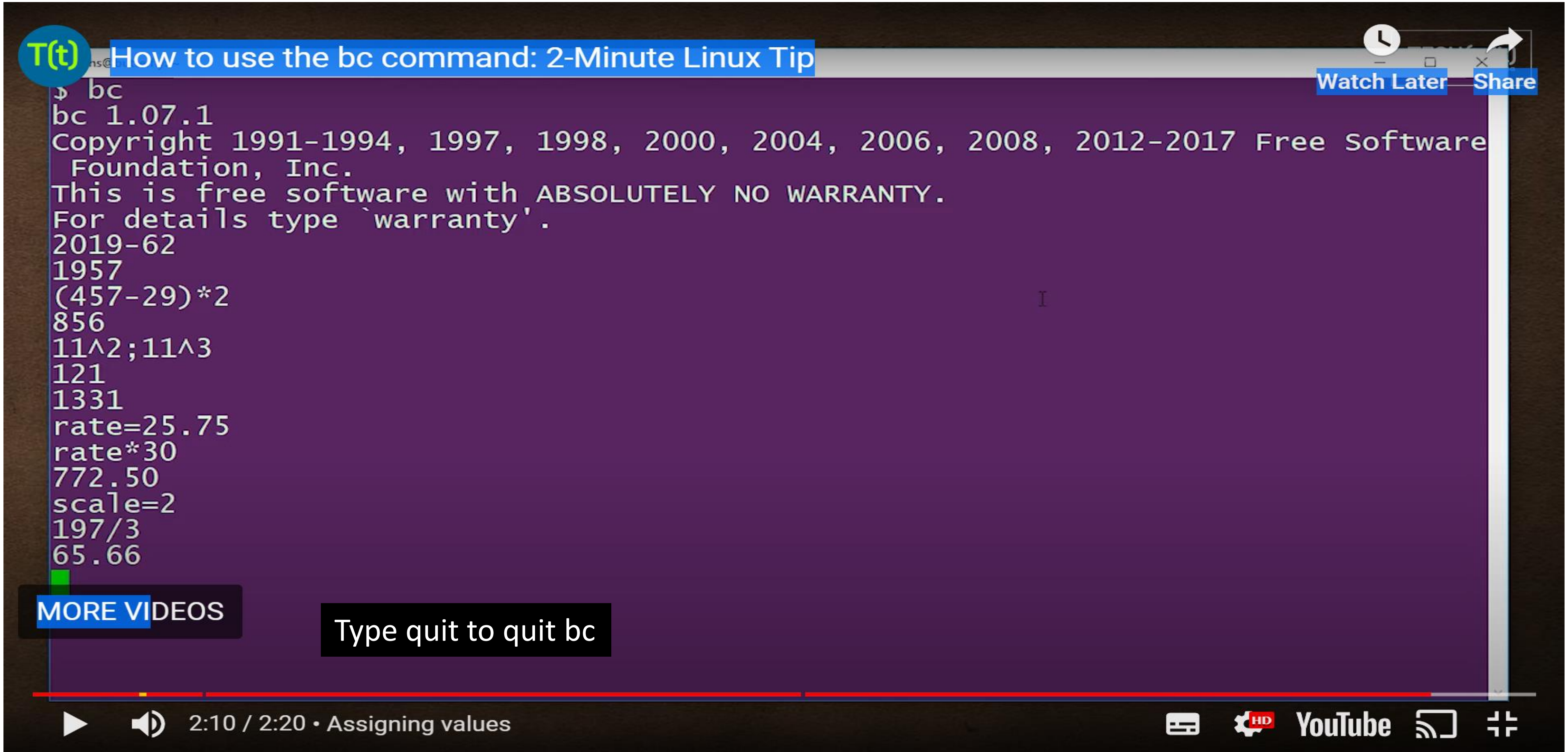
- to perform integer increment and decreament.
- exponent operations like **x^3**

bc command



```
linuxconfig.org:~$ echo '8.5 / 2.3' | bc
3

 linuxconfig.org:~$ echo 'scale=2;8.5 / 2.3' | bc
3.69

 linuxconfig.org:~$ echo 'scale=30;8.5 / 2.3' | bc
 3.695652173913043478260869565217

linuxconfig.org:~$ squareroot=$( echo 'scale=50;sqrt(50)' | bc )
 linuxconfig.org:~$ echo $squareroot
7.07106781186547524400844362104849039284835937688474

linuxconfig.org:~$
```

Another way to use bc – like a calculator



How to use the bc command: 2-Minute Linux Tip

```
$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software
 Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
2019-62
1957
(457-29)*2
856
11^2;11^3
121
1331
rate=25.75
rate*30
772.50
scale=2
197/3
65.66
```

Type quit to quit bc

2:10 / 2:20 • Assigning values