Robotics Corner

Robotics Corner

Linux

Linux is an open Source operating system that is widely used in most of the companies.

01 System Information

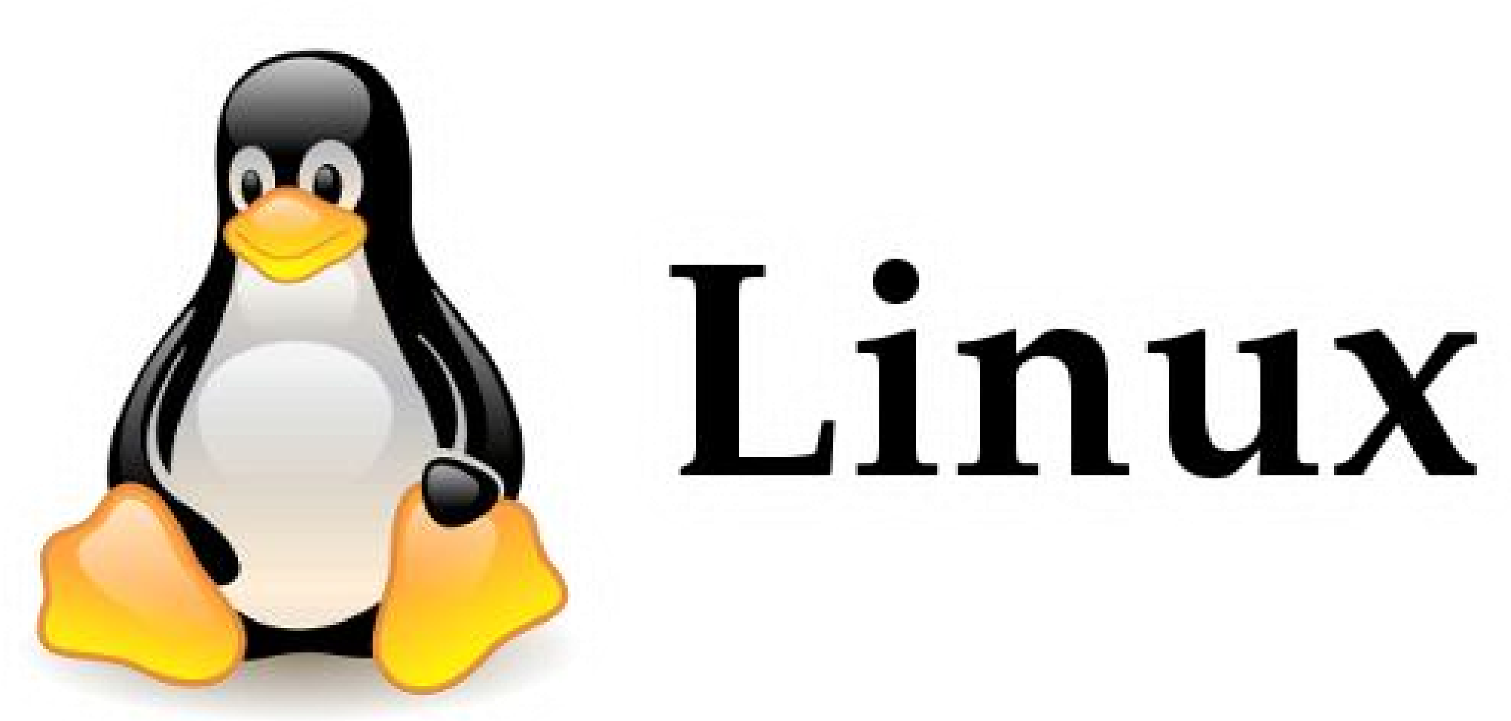
02 Managing Services

03 Package management

04 Basic shell scripting concepts

Linux

System Management and Basic Scripting



ROBOTICS CORNER

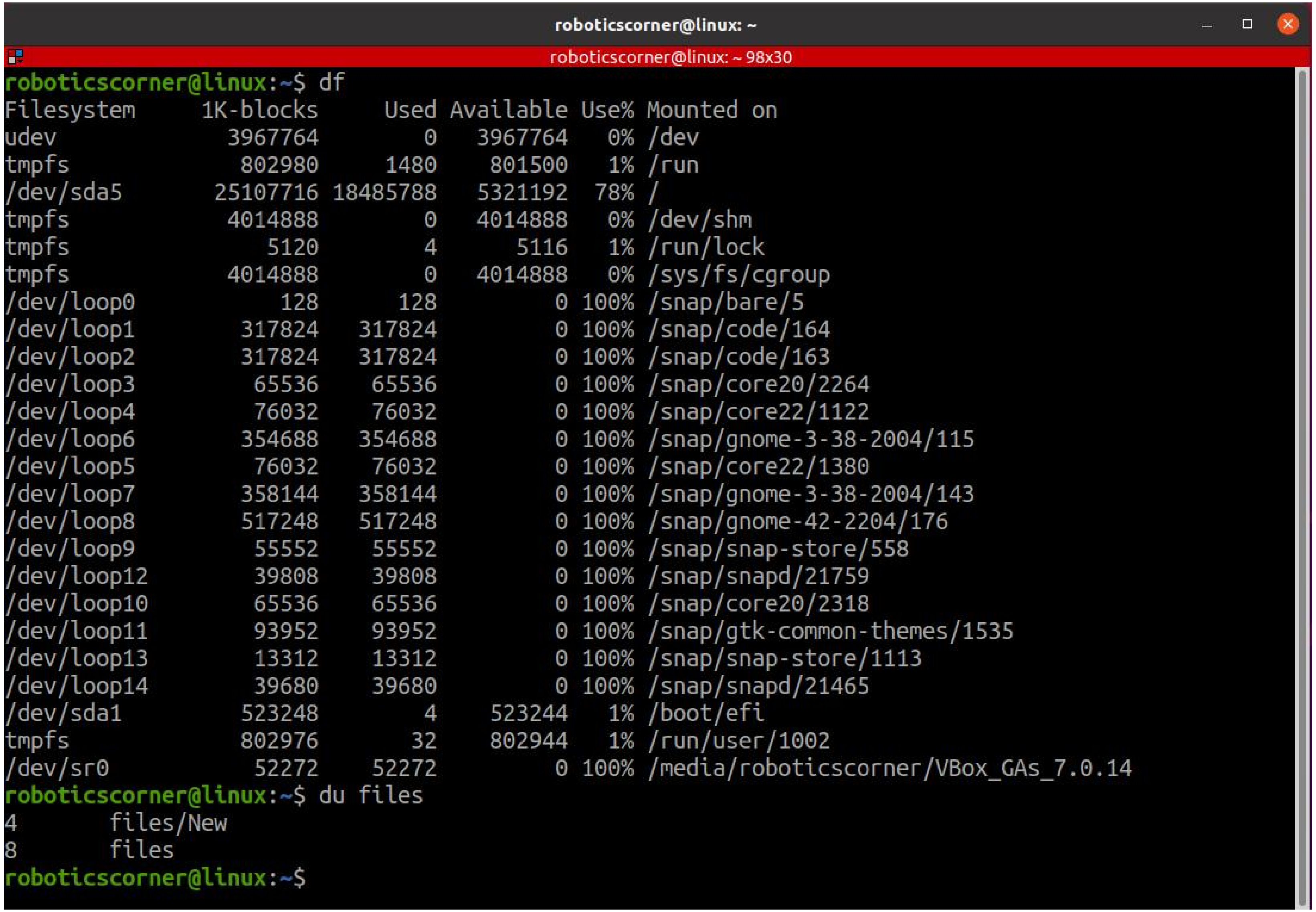
System information commands

- To know only the system name, you can use the uname command without any switch that will print system information or the uname -s command will print the kernel name of your system.
- To view your Linux network hostname, use the ‘-n’ switch with the uname command as shown.
- To get information about the Linux kernel version: uname -v
- To get the information about your Linux kernel release: uname -r
- To print your Linux hardware architecture name: uname -m
- All this information can be printed at once by running the ‘uname -a’ command

Df/du

- The "disk free" (df) command tells you the total disk size, space used, space available, usage percentage, and what partition the disk is mounted on.
- The "disk usage" (du) is used you need to see the size of a given directory or subdirectory. It runs at the object level and only reports on the specified stats at the time of execution.

01 System Information



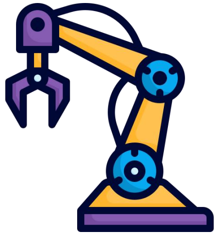
```
roboticscorner@linux:~$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
udev             3967764        0   3967764   0% /dev
tmpfs             802980     1480    801500   1% /run
/dev/sda5       25107716 18485788   5321192  78% /
tmpfs            4014888        0   4014888   0% /dev/shm
tmpfs               5120        4      5116   1% /run/lock
tmpfs            4014888        0   4014888   0% /sys/fs/cgroup
/dev/loop0           128      128         0 100% /snap/bare/5
/dev/loop1        317824   317824         0 100% /snap/code/164
/dev/loop2        317824   317824         0 100% /snap/code/163
/dev/loop3         65536    65536         0 100% /snap/core20/2264
/dev/loop4         76032    76032         0 100% /snap/core22/1122
/dev/loop6        354688   354688         0 100% /snap/gnome-3-38-2004/115
/dev/loop5         76032    76032         0 100% /snap/core22/1380
/dev/loop7        358144   358144         0 100% /snap/gnome-3-38-2004/143
/dev/loop8        517248   517248         0 100% /snap/gnome-42-2204/176
/dev/loop9         55552    55552         0 100% /snap/snap-store/558
/dev/loop12        39808    39808         0 100% /snap/snapd/21759
/dev/loop10        65536    65536         0 100% /snap/core20/2318
/dev/loop11        93952    93952         0 100% /snap/gtk-common-themes/1535
/dev/loop13        13312    13312         0 100% /snap/snap-store/1113
/dev/loop14        39680    39680         0 100% /snap/snapd/21465
/dev/sda1         523248        4    523244   1% /boot/efi
tmpfs             802976       32    802944   1% /run/user/1002
/dev/sr0           52272    52272         0 100% /media/roboticscorner/VBox_GAs_7.0.14
roboticscorner@linux:~$ du files
4       files/New
8       files
roboticscorner@linux:~$
```

Managing services and daemons

- An operating system requires programs that run in the background called services. In a Linux system, these services are called daemons. They are managed using an init system like systemd.
- In Unix-based computer operating systems, init (short for initialization) is the first process started during booting of the operating system. Init is a daemon process that continues running until the system is shut down.

Process vs service

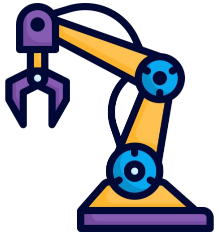
- A process is an instance of a running program. When you execute a program, it becomes a process.
- Processes are the basic units of execution in a Linux system.
- Each process has a unique process ID (PID) assigned to it.
- Processes have their own memory space, file descriptors, and other resources.



Process vs service

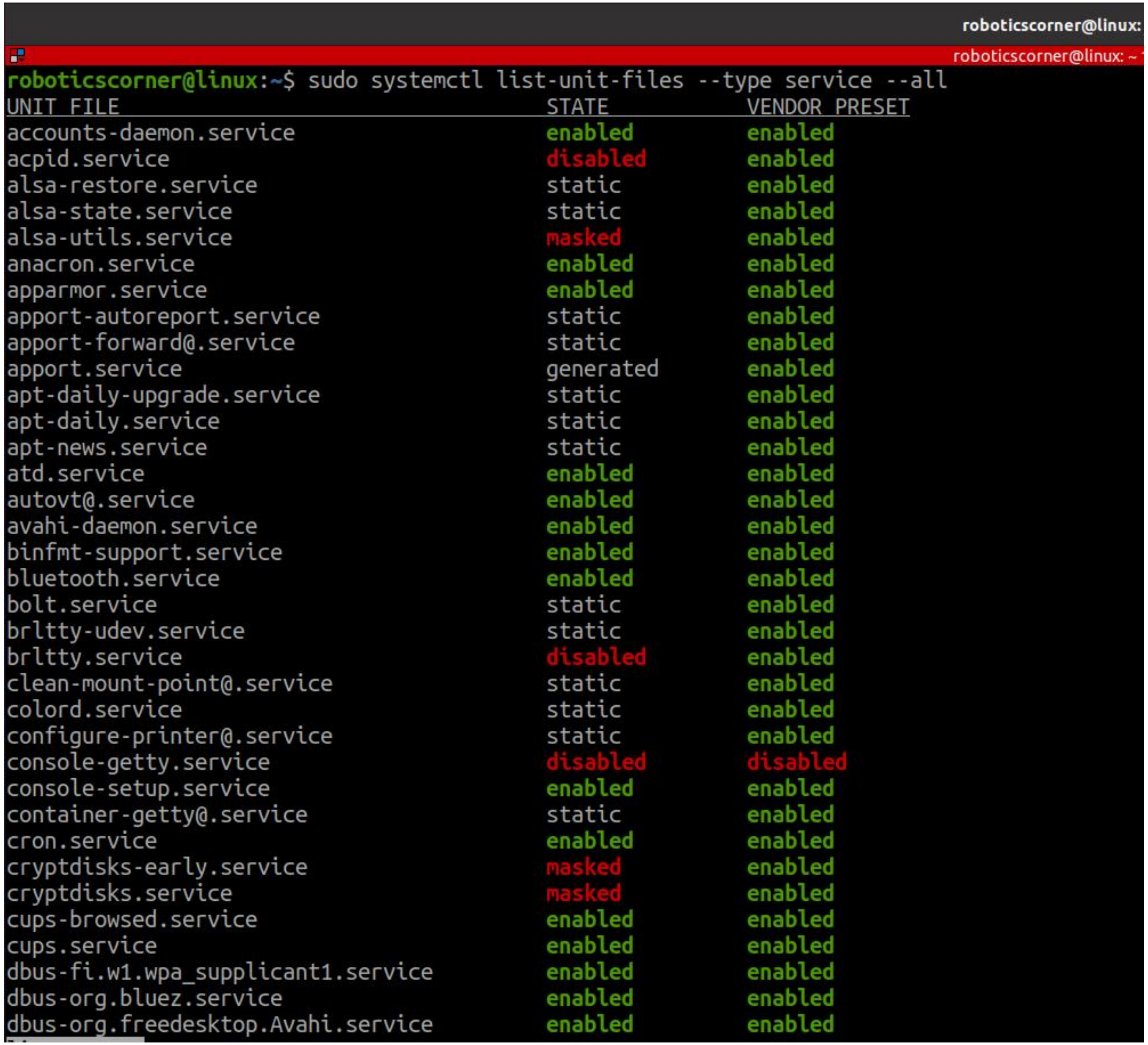
- A service is a background process or daemon that runs on a system to provide specific functionality or perform specific tasks.
- Services often start when the system boots up and continue running in the background, waiting for specific events or requests.
- Services are usually managed by an init system like systemd, which can start, stop, restart, and manage their lifecycle, there are others, but system is the most used.

Managing services and daemons



- Most modern Linux systems use systemd – an init system and service manager for controlling daemons. It is a drop-in replacement for older distributions' init processes: pstree → started by systemd
- Systemd has the systemctl command, which lets users manage their system and service configurations. For example, use it to list all unit files in your Linux server: daemons = units = services
- sudo systemctl list-unit-files --type service --all

02 Managing Services



```
roboticscorner@linux:~$ sudo systemctl list-unit-files --type service --all
UNIT FILE                                  STATE           VENDOR PRESET
accounts-daemon.service                    enabled         enabled
acpid.service                              disabled        enabled
alsa-restore.service                       static          enabled
alsa-state.service                         static          enabled
alsa-utils.service                         masked          enabled
anacron.service                            enabled         enabled
apparmor.service                           enabled         enabled
apport-autoreport.service                  static          enabled
apport-forward@.service                    static          enabled
apport.service                             generated       enabled
apt-daily-upgrade.service                  static          enabled
apt-daily.service                          static          enabled
apt-news.service                           static          enabled
atd.service                                enabled         enabled
autovt@.service                            enabled         enabled
avahi-daemon.service                       enabled         enabled
binfmt-support.service                     enabled         enabled
bluetooth.service                          enabled         enabled
bolt.service                               static          enabled
brltty-udev.service                        static          enabled
brltty.service                             disabled        enabled
clean-mount-point@.service                 static          enabled
colord.service                             static          enabled
configure-printer@.service                 static          enabled
console-getty.service                      disabled        disabled
console-setup.service                      enabled         enabled
container-getty@.service                   static          enabled
cron.service                               enabled         enabled
cryptdisks-early.service                   masked          enabled
cryptdisks.service                         masked          enabled
cups-browsed.service                       enabled         enabled
cups.service                               enabled         enabled
dbus-fi.w1.wpa_supplicant1.service         enabled         enabled
dbus-org.bluez.service                     enabled         enabled
dbus-org.freedesktop.Avahi.service         enabled         enabled
```

sudo systemctl list-unit-files --type service --all

- Enabled – active services running in the background.
- Disabled – disabled services that users can enable directly using the start command.
- Masked – stopped services that can only be started by removing the masked property.
- Static – services that only run when another program or unit requires them.
- Failed – inactive services that can’t load or operate properly.

Managing services and daemons

- To stop a service: sudo systmctl stop [SERVICE]
- To get the status of a service: sudo systmctl status [SERVICE]
- To start a service: sudo systmctl start [SERVICE]
- To restart a service: sudo systmctl restart [SERVICE]

03 Package Management

- When it comes to package management on Linux systems, two popular tools are YUM and APT. YUM, short for Yellowdog Updater Modified, is commonly used in Red Hat-based distributions like CentOS and RHEL. On the other hand, APT, which stands for Advanced Packaging Tool, is widely used in Debian, Ubuntu, and their derivatives. Understanding the differences between these package managers can help you effectively manage software installations and updates on your Linux system.

YUM vs. APT: Managing Software Packages on Linux

- YUM and APT are package managers that simplify the installation, upgrade, and configuration of software packages on Linux systems. While they serve the same purpose, there are some notable differences between them.

APT

- APT uses .deb files as the package format and is primarily used in Debian, Ubuntu, and related distributions.
- APT provides several commonly used commands, such as update, upgrade, install, remove, purge, list, and search.
- APT organizes options into functional groups and stores them in the /etc/apt/apt.conf file, which is organized in a tree structure.

apt

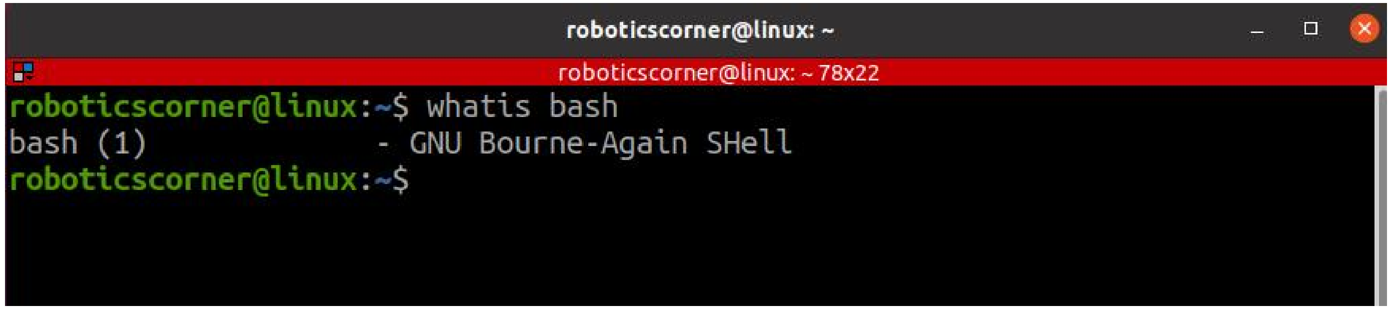
- When you install an application using apt in Ubuntu, the package manager (apt) does not contain all the data for the application itself. Instead, apt relies on repositories, which are online databases of software packages maintained by Ubuntu and its community.
- When you instruct apt to install a package, it searches the repositories configured on your system to find the package and its dependencies. If the package and its dependencies are found, apt downloads them from the repository over the internet and installs them on your system.
- The repositories contain metadata about the packages, such as their names, versions, descriptions, dependencies, and download locations. apt uses this metadata to locate and install the requested packages.
- So, while apt itself doesn't contain all the data for each application, it acts as a tool to efficiently manage the installation and removal of software packages from online repositories.

yum

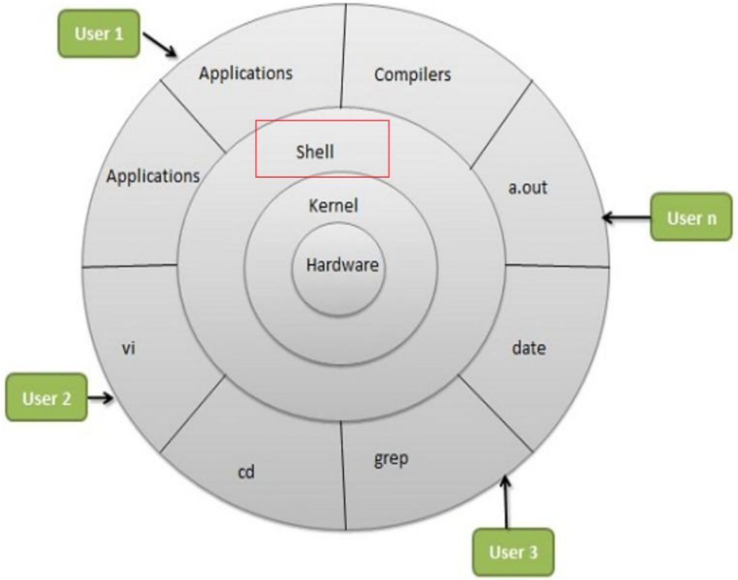
- YUM uses .rpm files and is commonly used in Red Hat-based distributions like CentOS, RHEL, Fedora, and OpenSUSE.
- YUM offers commands like install, remove, search, info, and update.
- YUM allows options to be set with global and repository-specific effects, and the configuration is managed in the /etc/yum.conf file

Basic shell scripting concepts

- Shell is an interpreter for the applications/commands of a user to make the kernel manage the hardware.
- It stands for Bourne-Again SHell



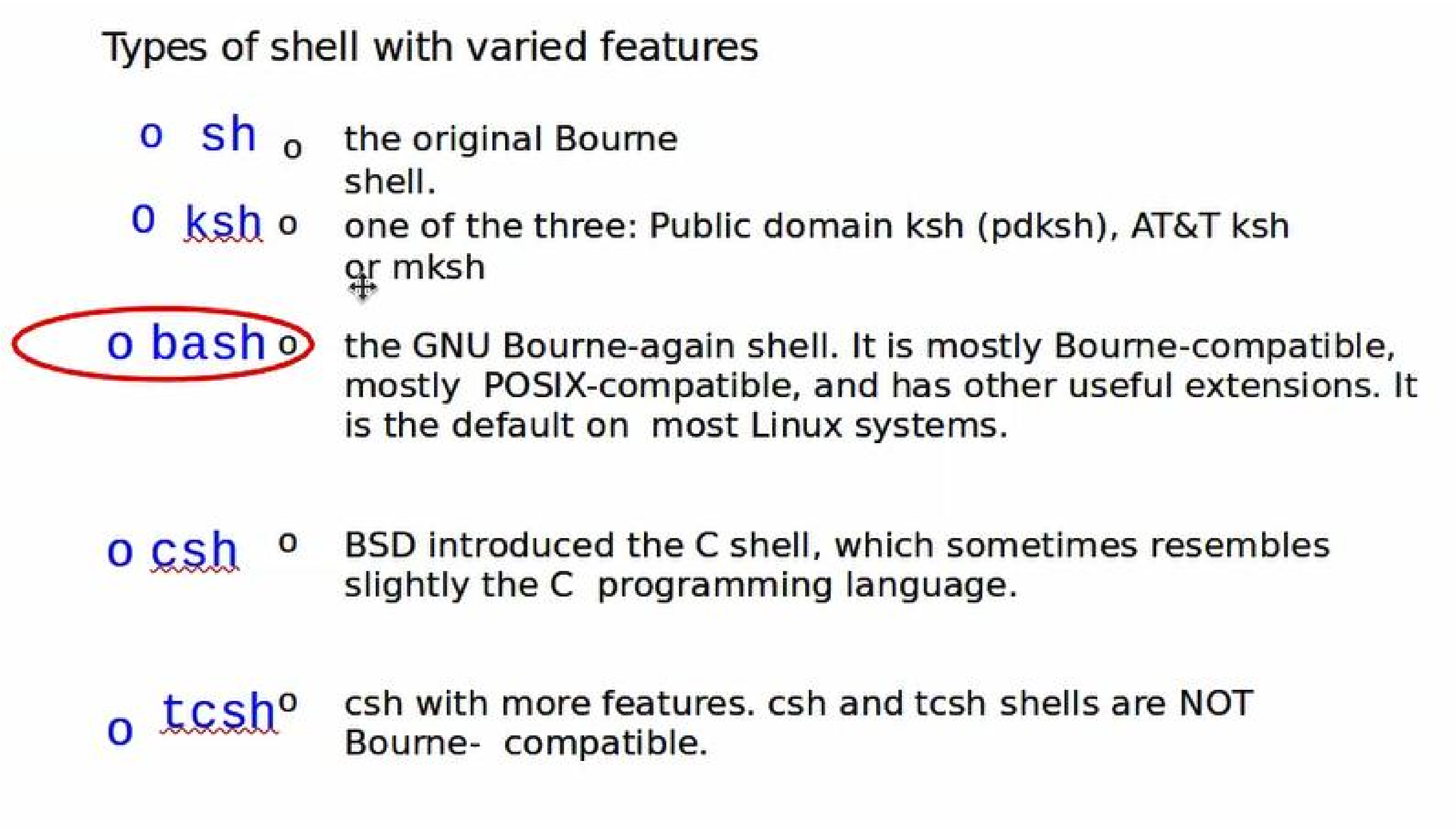
```
roboticscorner@linux:~$ whatis bash
bash (1)             - GNU Bourne-Again SHell
roboticscorner@linux:~$
```



04

Basic shell scripting concepts

Types of shell



Types of shell with varied features

o sh o the original Bourne shell.

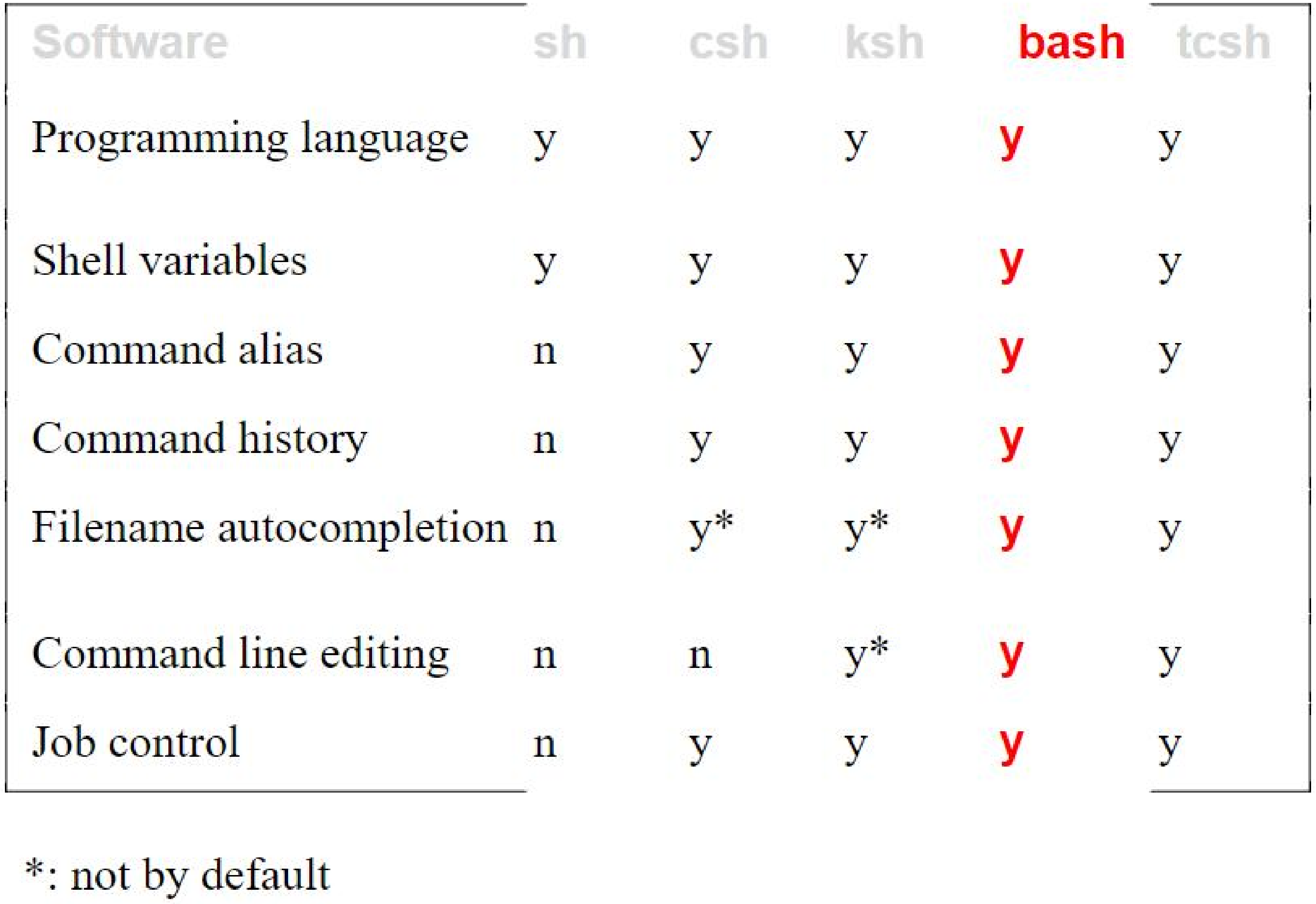
o ksh o one of the three: Public domain ksh (pdksh), AT&T ksh or mksh

o bash o the GNU Bourne-again shell. It is mostly Bourne-compatible, mostly POSIX-compatible, and has other useful extensions. It is the default on most Linux systems.

o csh o BSD introduced the C shell, which sometimes resembles slightly the C programming language.

o tcsh o csh with more features. csh and tcsh shells are NOT Bourne- compatible.

Shell comparison



Software	sh	csh	ksh	bash	tcsh
Programming language	y	y	y	y	y
Shell variables	y	y	y	y	y
Command alias	n	y	y	y	y
Command history	n	y	y	y	y
Filename autocompletion	n	y*	y*	y	y
Command line editing	n	n	y*	y	y
Job control	n	y	y	y	y

*: not by default

- Bash supports all because it was created by the linux foundation(non-profit organization) →time and resources are abundant

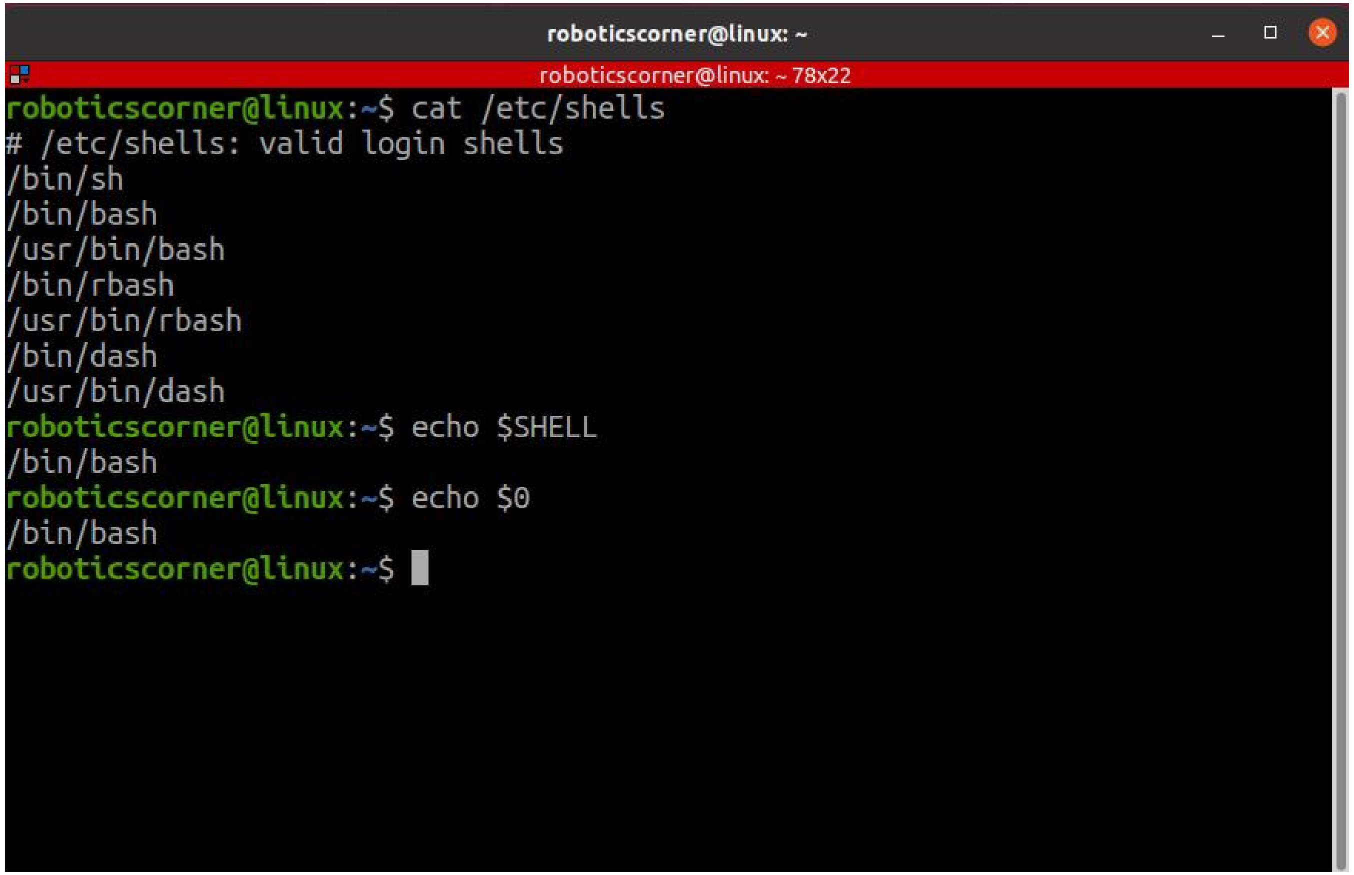
What can you do with a shell

- File Management, Directory Management
- Process Management, compile and run applications
- Network Management
- Shell Scripting

04

Basic shell scripting concepts

- List available shells on the system: cat /etc/shells
- To check the current shell you are using: echo $0 or echo $SHELL



```
roboticscorner@linux:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
roboticscorner@linux:~$ echo $SHELL
/bin/bash
roboticscorner@linux:~$ echo $0
/bin/bash
roboticscorner@linux:~$ 
```

Shell scripting

- Script: a program written for a software environment to automate execution of tasks
- A series of shell commands put together in a file
- When the script is executed, those commands will be executed one line at a time automatically
- Shell script is interpreted, not compiled.

When not to use shell scripting

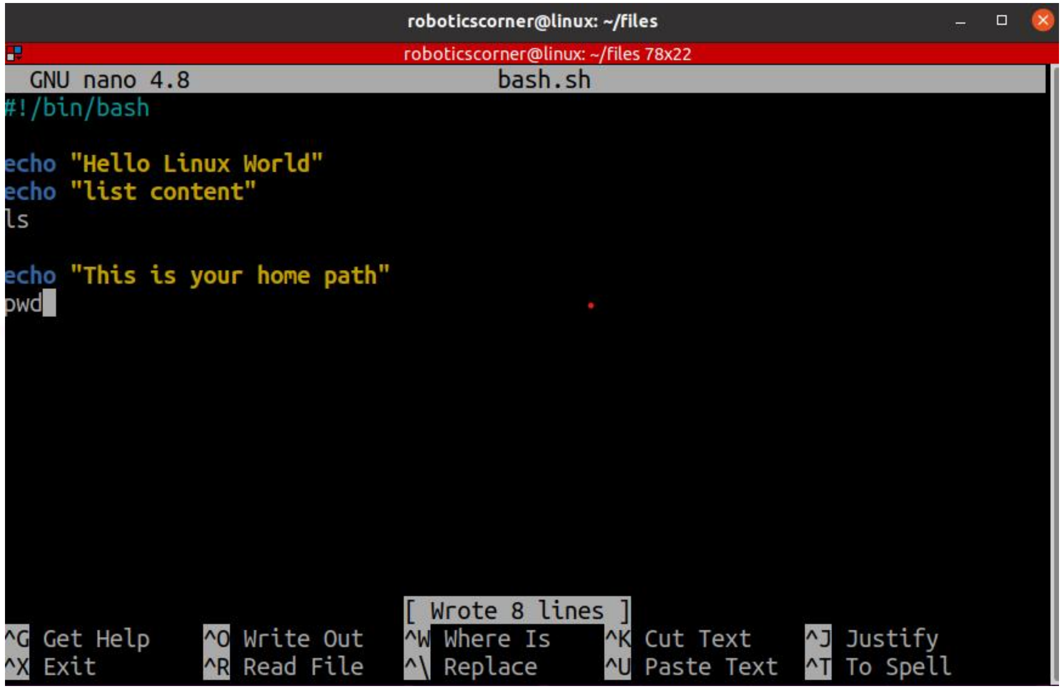
- Performance/Security-Critical Applications
- Complex Data Structures and Algorithms
- Cross-Platform Development
- Large Software Projects

Hello

```
#!/bin/bash
 # Ascript example
echo "Hello World!" # print something
```

1. #!: "Shebang" line to instruct which interpreter to use. In the current example, bash. For tcsh, it would be: #!/ bin/tcsh

1. All comments begin with "#".
2. Print "Hello World!" to the screen.


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 78x22

```
  GNU nano 4.8                        bash.sh
#!/bin/bash

echo "Hello Linux World"
echo "list content"
ls

echo "This is your home path"
pwd
















                           [ Wrote 8 lines ]
^G Get Help     ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify
^X Exit         ^R Read File    ^\ Replace      ^U Paste Text   ^T To Spell
```

```
$ ./hello_world.sh # using default  Hello World!
$ bash hello_world.sh # using bash script to run the /bin/bash
Hello World!
```

Hello

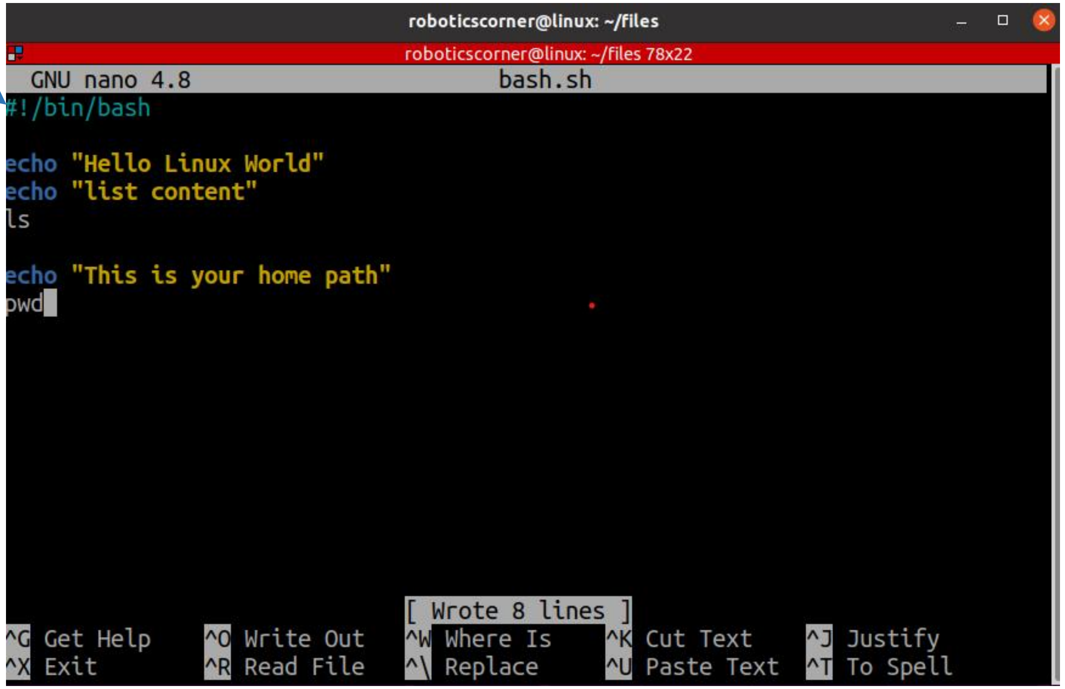
Do not forget the shebang line

```
#!/bin/bash
 # Ascript example
echo "Hello World!" # print something
```

1. `#!:` "Shebang" line to instruct which interpreter to use. In the current example, bash. For tcsh, it would be: `#!/bin/tcsh`

1. All comments begin with "#".
2. Print "Hello World!" to the screen.

```
$ ./hello_world.sh # using default  Hello World!
$ bash hello_world.sh # using bash script to run the /bin/bash
Hello World!
```



roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 78x22
GNU nano 4.8
bash.sh

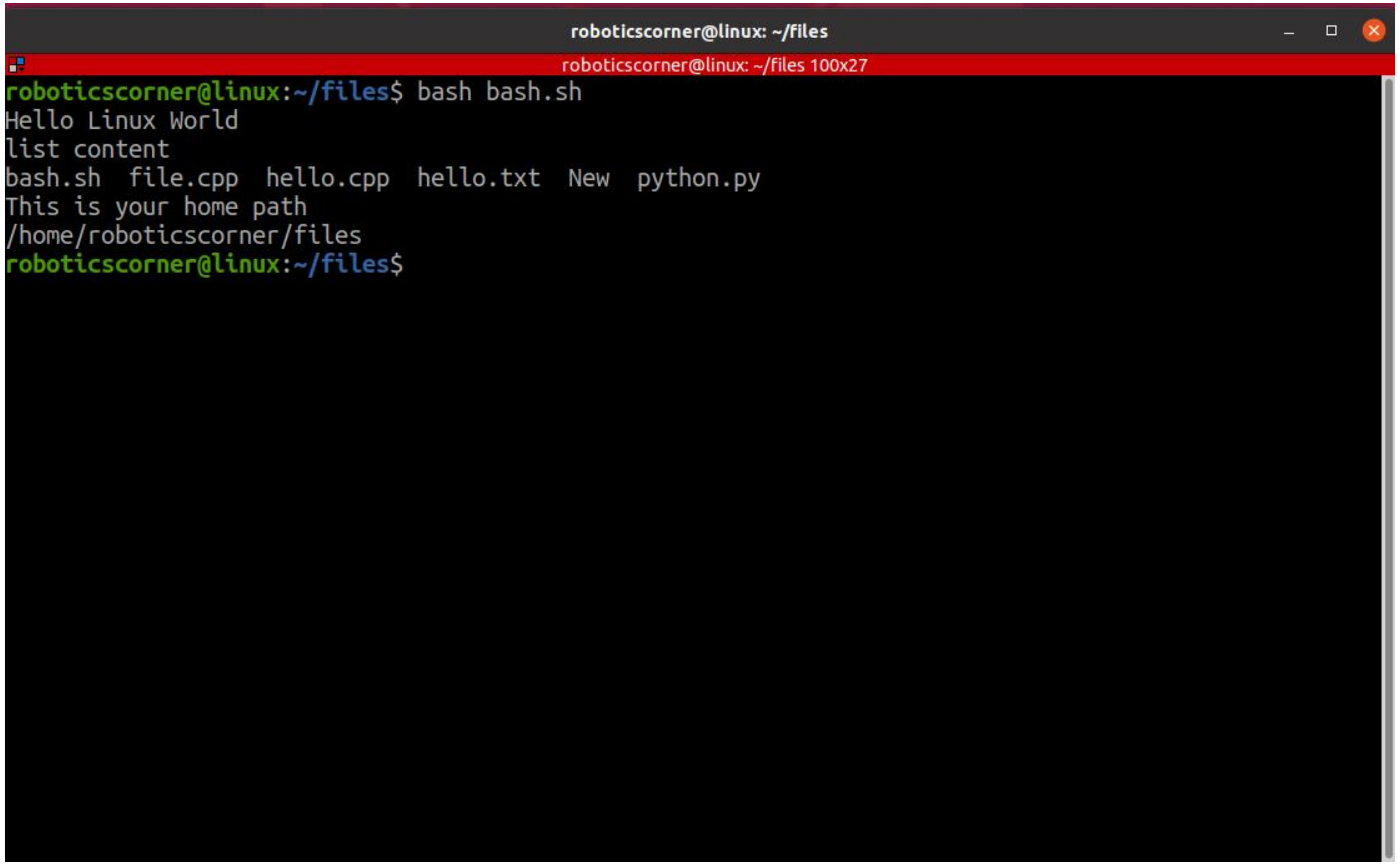
```
#!/bin/bash

echo "Hello Linux World"
echo "list content"
ls

echo "This is your home path"
pwd
```

```
[ Wrote 8 lines ]
```

How to run the bash file

There is 2 ways

But! That’s not actually how we run it.


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 100x27

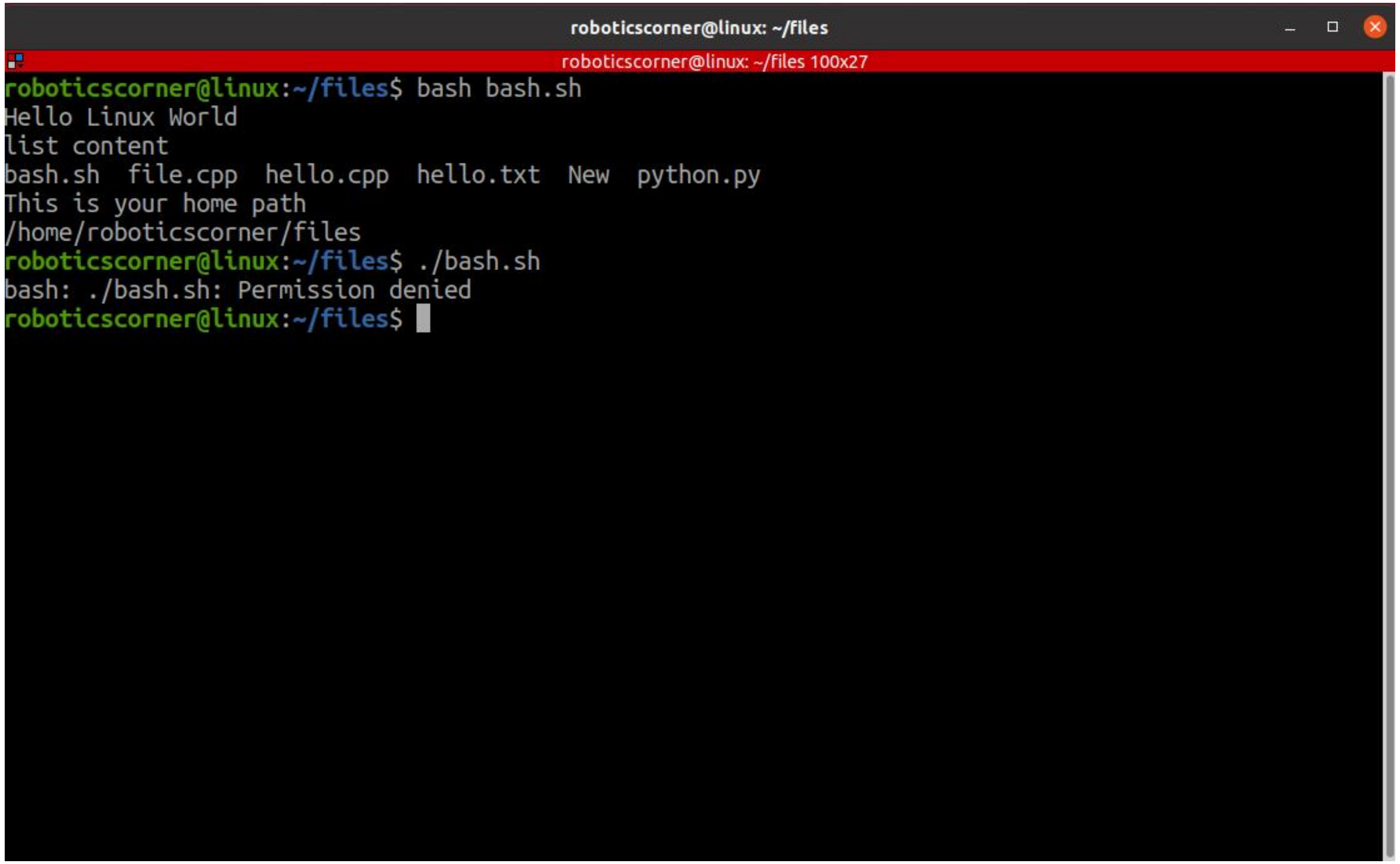
```
roboticscorner@linux:~/files$ bash bash.sh
Hello Linux World
list content
bash.sh  file.cpp  hello.cpp  hello.txt  New  python.py
This is your home path
/home/roboticscorner/files
roboticscorner@linux:~/files$
```

How to run the bash file

There is 2 ways

But! That’s not actually how we run it.

We have to make it executable


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 100x27

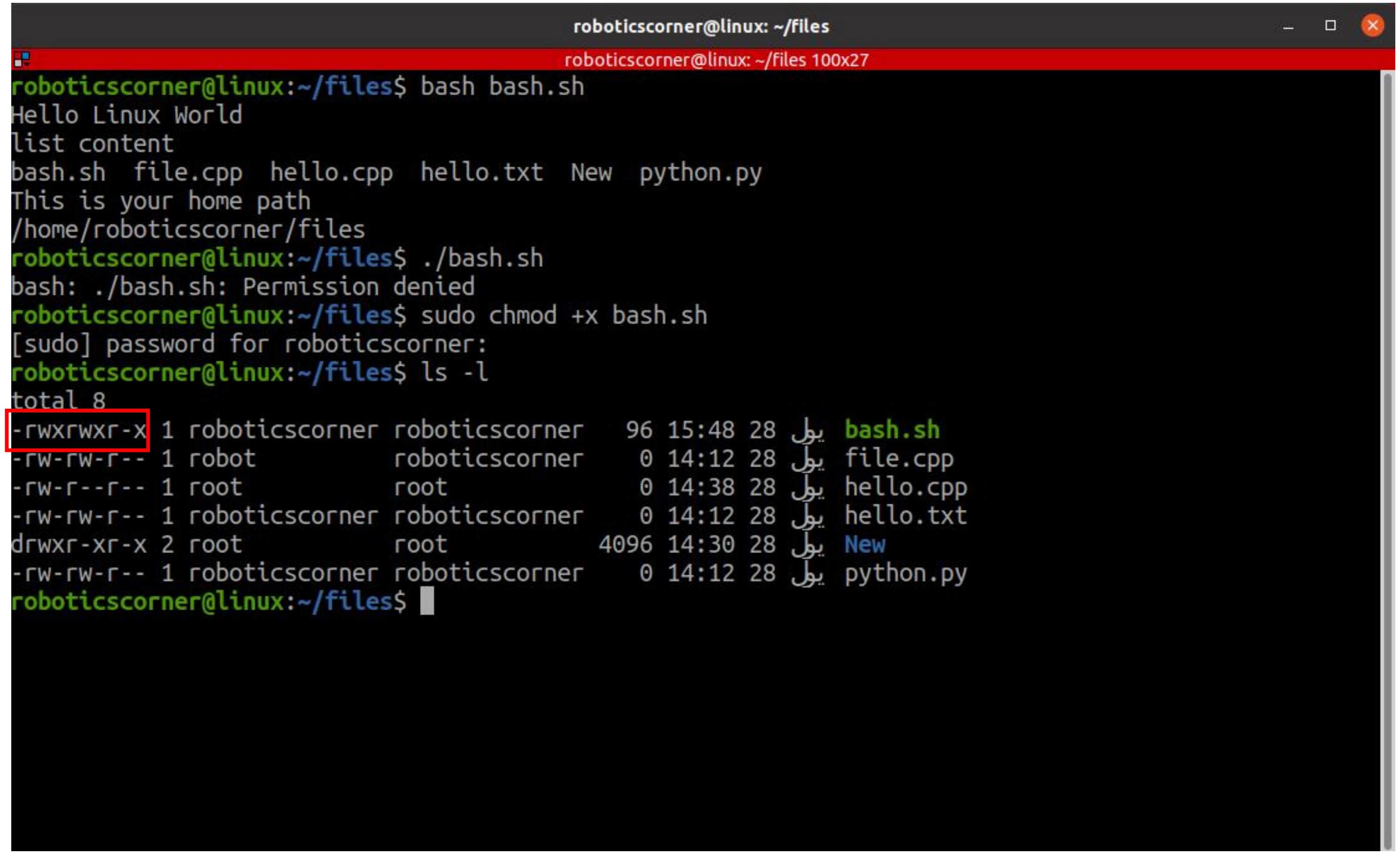
```
roboticscorner@linux:~/files$ bash bash.sh
Hello Linux World
list content
bash.sh  file.cpp  hello.cpp  hello.txt  New  python.py
This is your home path
/home/roboticscorner/files
roboticscorner@linux:~/files$ ./bash.sh
bash: ./bash.sh: Permission denied
roboticscorner@linux:~/files$ 
```

How to run the bash file

There is 2 ways

But! That’s not actually how we run it.

We have to make it executable


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 100x27

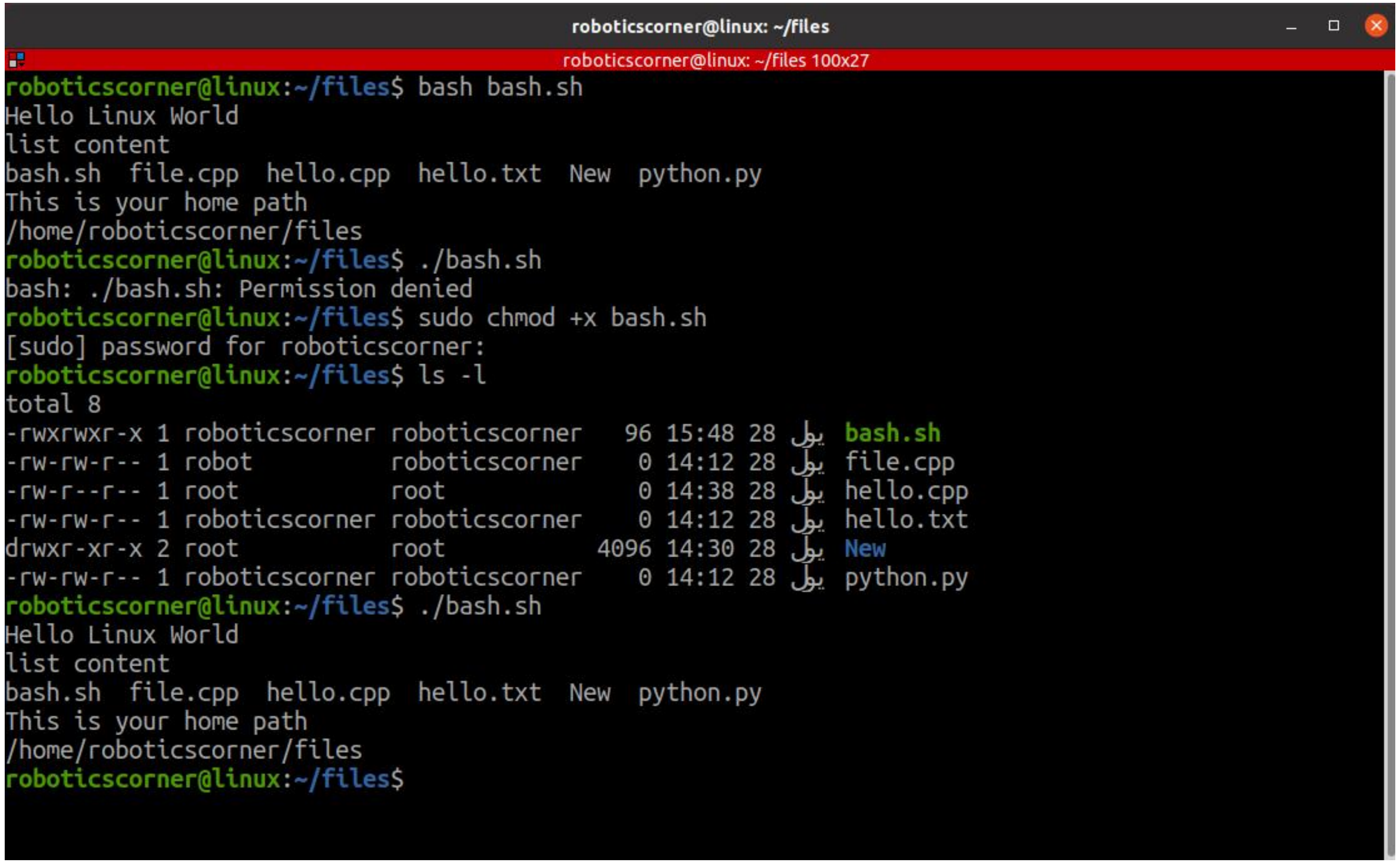
```
roboticscorner@linux:~/files$ bash bash.sh
Hello Linux World
list content
bash.sh  file.cpp  hello.cpp  hello.txt  New  python.py
This is your home path
/home/roboticscorner/files
roboticscorner@linux:~/files$ ./bash.sh
bash: ./bash.sh: Permission denied
roboticscorner@linux:~/files$ sudo chmod +x bash.sh
[sudo] password for roboticscorner:
roboticscorner@linux:~/files$ ls -l
total 8
-rwxrwxr-x 1 roboticscorner roboticscorner   96 15:48 28 يول bash.sh
-rw-rw-r-- 1 robot          roboticscorner    0 14:12 28 يول file.cpp
-rw-r--r-- 1 root           root              0 14:38 28 يول hello.cpp
-rw-rw-r-- 1 roboticscorner roboticscorner    0 14:12 28 يول hello.txt
drwxr-xr-x 2 root           root           4096 14:30 28 يول New
-rw-rw-r-- 1 roboticscorner roboticscorner    0 14:12 28 يول python.py
roboticscorner@linux:~/files$ 
```

How to run the bash file

There is 2 ways

But! That’s not actually how we run it.

We have to make it executable


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 100x27

```
roboticscorner@linux:~/files$ bash bash.sh
Hello Linux World
list content
bash.sh  file.cpp  hello.cpp  hello.txt  New  python.py
This is your home path
/home/roboticscorner/files
roboticscorner@linux:~/files$ ./bash.sh
bash: ./bash.sh: Permission denied
roboticscorner@linux:~/files$ sudo chmod +x bash.sh
[sudo] password for roboticscorner:
roboticscorner@linux:~/files$ ls -l
total 8
-rwxrwxr-x 1 roboticscorner roboticscorner   96 15:48 28 يول bash.sh
-rw-rw-r-- 1 robot          roboticscorner    0 14:12 28 يول file.cpp
-rw-r--r-- 1 root           root              0 14:38 28 يول hello.cpp
-rw-rw-r-- 1 roboticscorner roboticscorner    0 14:12 28 يول hello.txt
drwxr-xr-x 2 root           root           4096 14:30 28 يول New
-rw-rw-r-- 1 roboticscorner roboticscorner    0 14:12 28 يول python.py
roboticscorner@linux:~/files$ ./bash.sh
Hello Linux World
list content
bash.sh  file.cpp  hello.cpp  hello.txt  New  python.py
This is your home path
/home/roboticscorner/files
roboticscorner@linux:~/files$
```

Interactive vs noninteractive shell

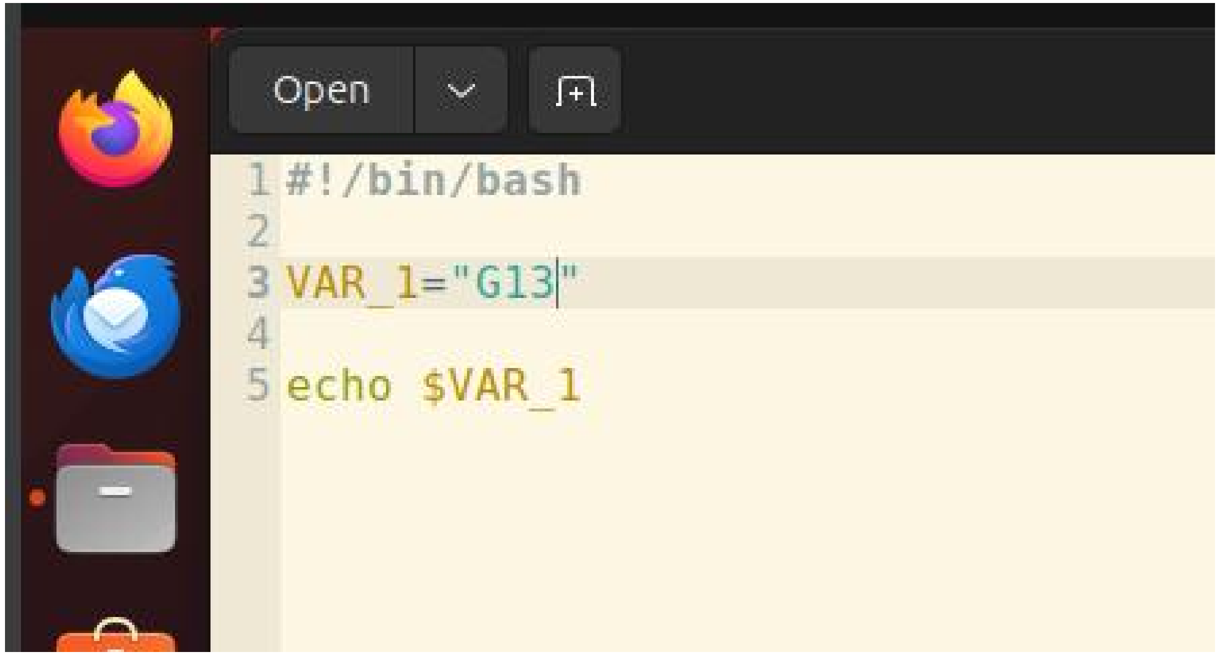
- An interactive shell refers to a command-line interface that allows users to interact with the computer's operating system by typing commands in real-time and receiving immediate feedback. In an interactive shell, users can enter commands, execute programs, and perform various tasks by typing text-based commands.
- A non-interactive shell is a shell session that runs without direct interaction with a user through a command-line interface. In a non-interactive shell, commands are often executed from scripts or other automated processes, and there is typically no user input or interaction during the execution. A shell running a script is always a non-interactive shell.

subshell

- A subshell is a child shell that is spawned by the main shell (also known as the parent shell). It is a separate process with its own set of variables and command history, and it allows you to execute commands and perform operations within a separate environment.

Variables in shell scripting

- A symbolic name for a chunk of memory to which we can assign values, read and manipulate its contents.



```
#!/bin/bash

VAR_1="G13"

echo $VAR_1
```

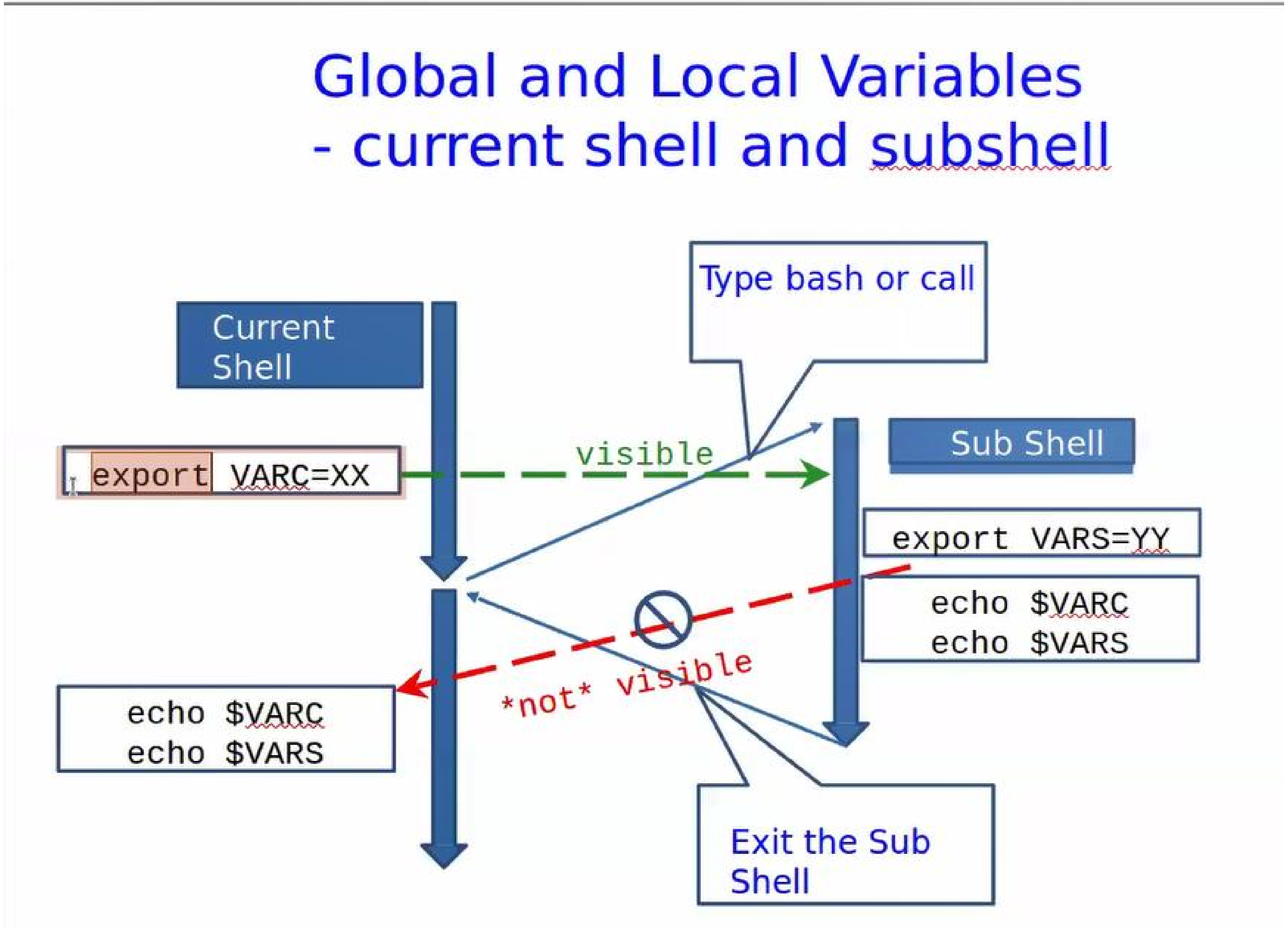
Variables

- Must start with a letter or underscore
- Number can be used anywhere else
- Do not use special characters such as @,#,%,$
- Case sensitive
- Allowed: VARIABLE, VAR1234able, var_name, _VAR
- Not allowed: 1var, %name, $myvar, var@NAME, myvar-1
- To reference a variable, prepend $ to the name of the variable
- Example: $PATH, $LD_LIBRARY_PATH, $myvar etc.

Local and global variables

- Variables created and used inside a shell are only local to the shell and can’t be seen outside this shell.
- If a variable is created in the terminal and used in a process for example, this variable is local to this terminal and can’t be seen elsewhere.
- Global variables are variables that are accessible and can be modified throughout the entire script, regardless of their initial declaration.
- Like: PATH, LD_LIBRARY_PATH, DISPLAY
- cmake –version: export name=value(directory)

04 Basic shell scripting concepts



Global and Local Variables - current shell and subshell

```
export VARC=XX
```

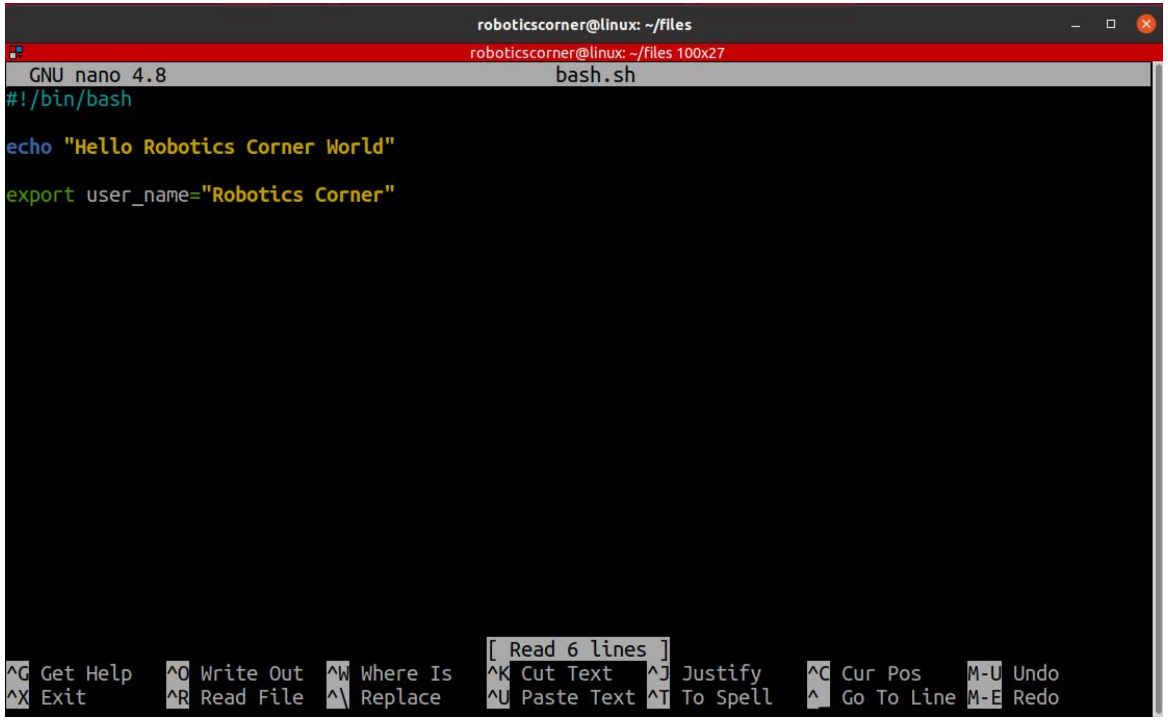
```
export VARS=YY
```

```
echo $VARC
echo $VARS
```

```
echo $VARC
echo $VARS
```

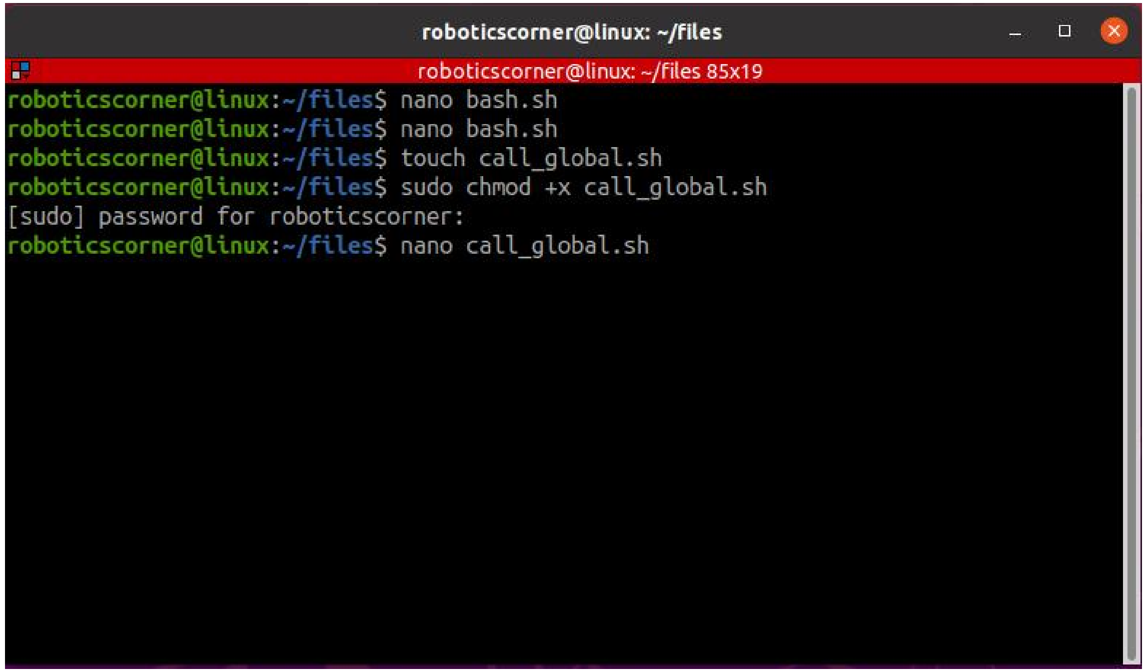
04

Basic shell scripting concepts



roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 100x27

```
  GNU nano 4.8                              bash.sh
#!/bin/bash

echo "Hello Robotics Corner World"

export user_name="Robotics Corner"

                                          [ Read 6 lines ]
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo
^X Exit       ^R Read File  ^\ Replace    ^U Paste Text ^T To Spell   ^_ Go To Line M-E Redo
```



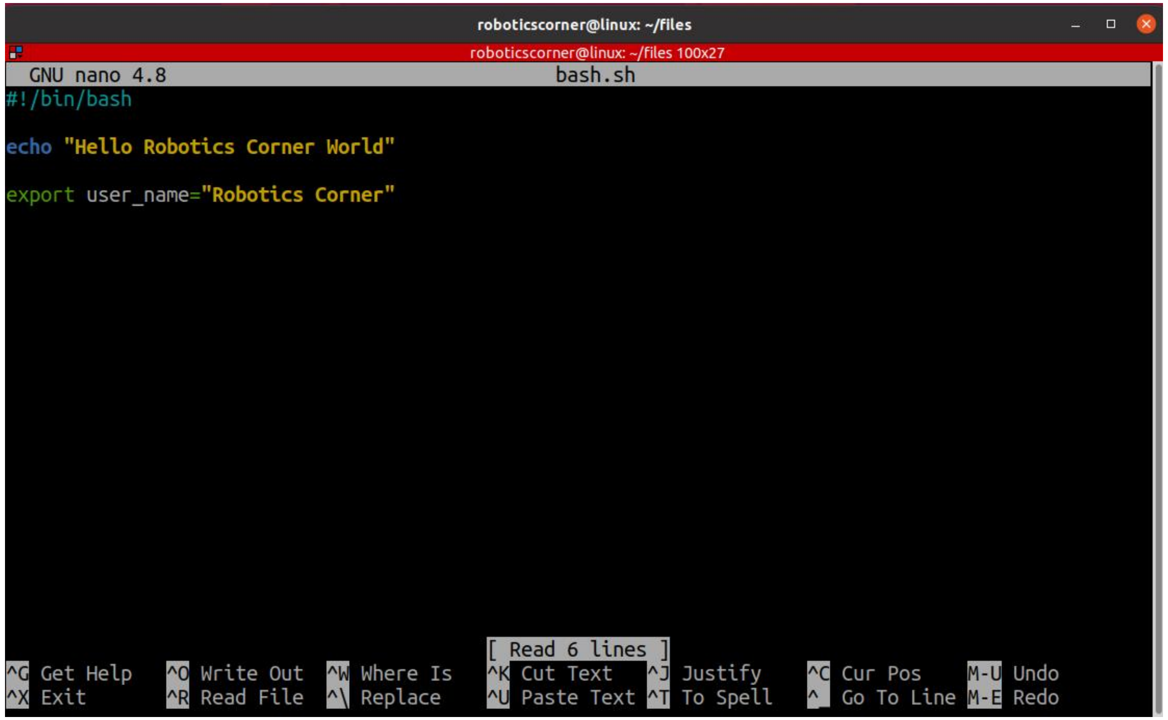
roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 85x19

```
roboticscorner@linux:~/files$ nano bash.sh
roboticscorner@linux:~/files$ nano bash.sh
roboticscorner@linux:~/files$ touch call_global.sh
roboticscorner@linux:~/files$ sudo chmod +x call_global.sh
[sudo] password for roboticscorner:
roboticscorner@linux:~/files$ nano call_global.sh
```

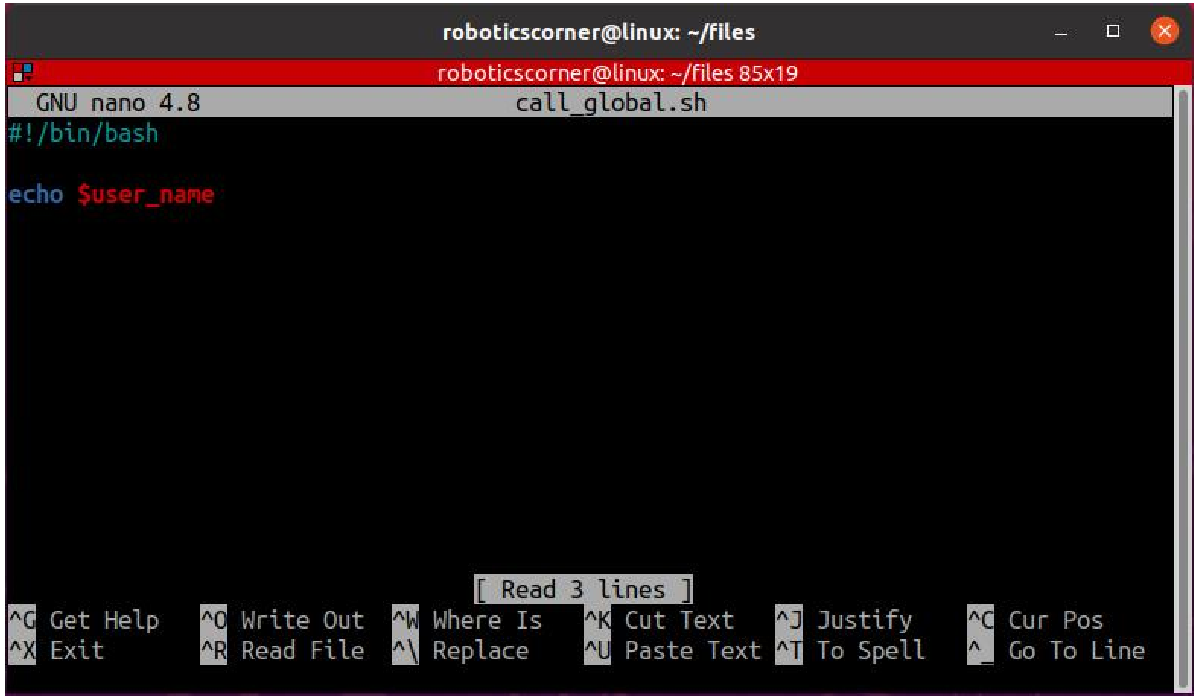
ROBOTICS CORNER

04

Basic shell scripting concepts

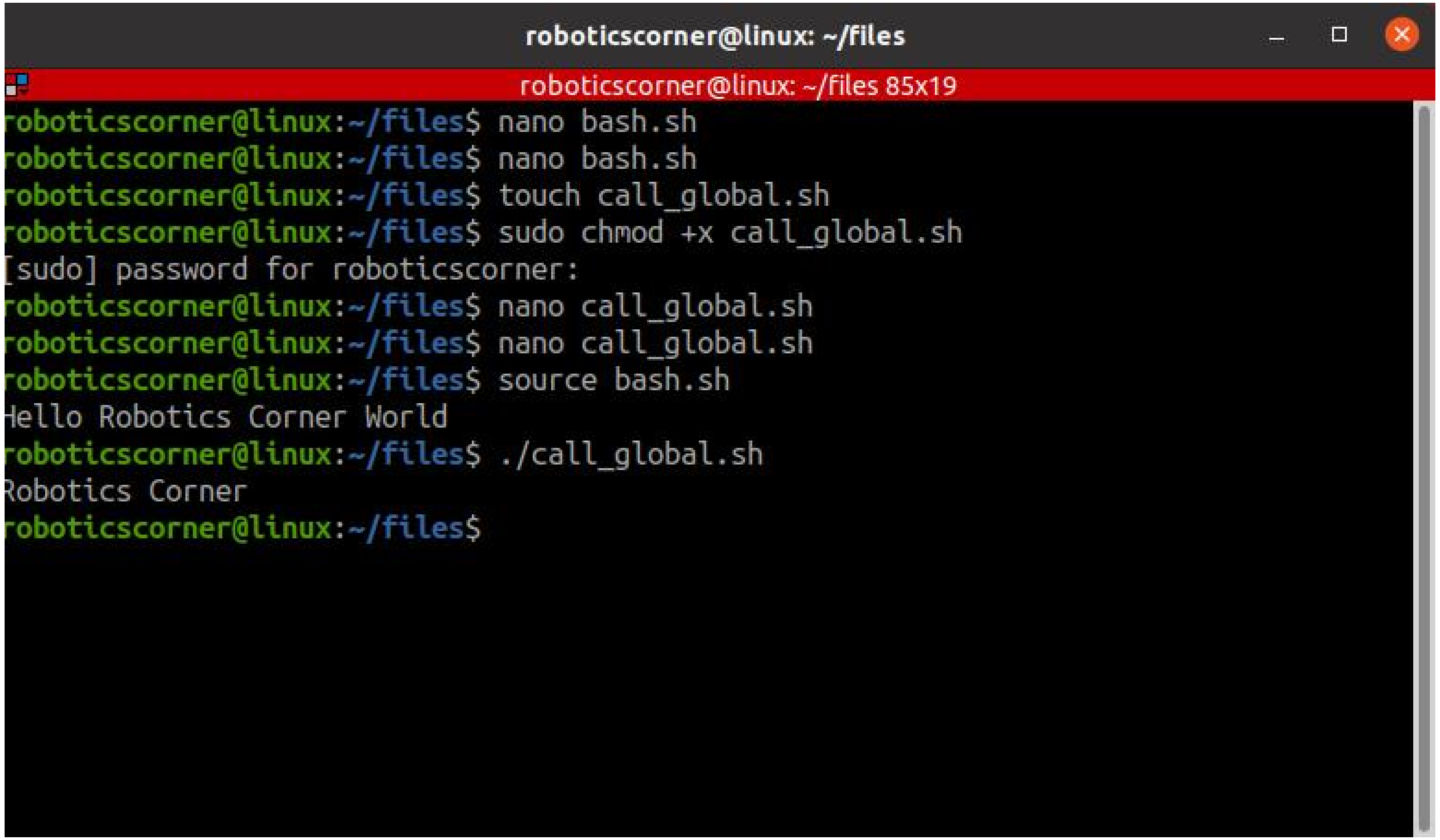


```
#!/bin/bash

echo "Hello Robotics Corner World"

export user_name="Robotics Corner"
```



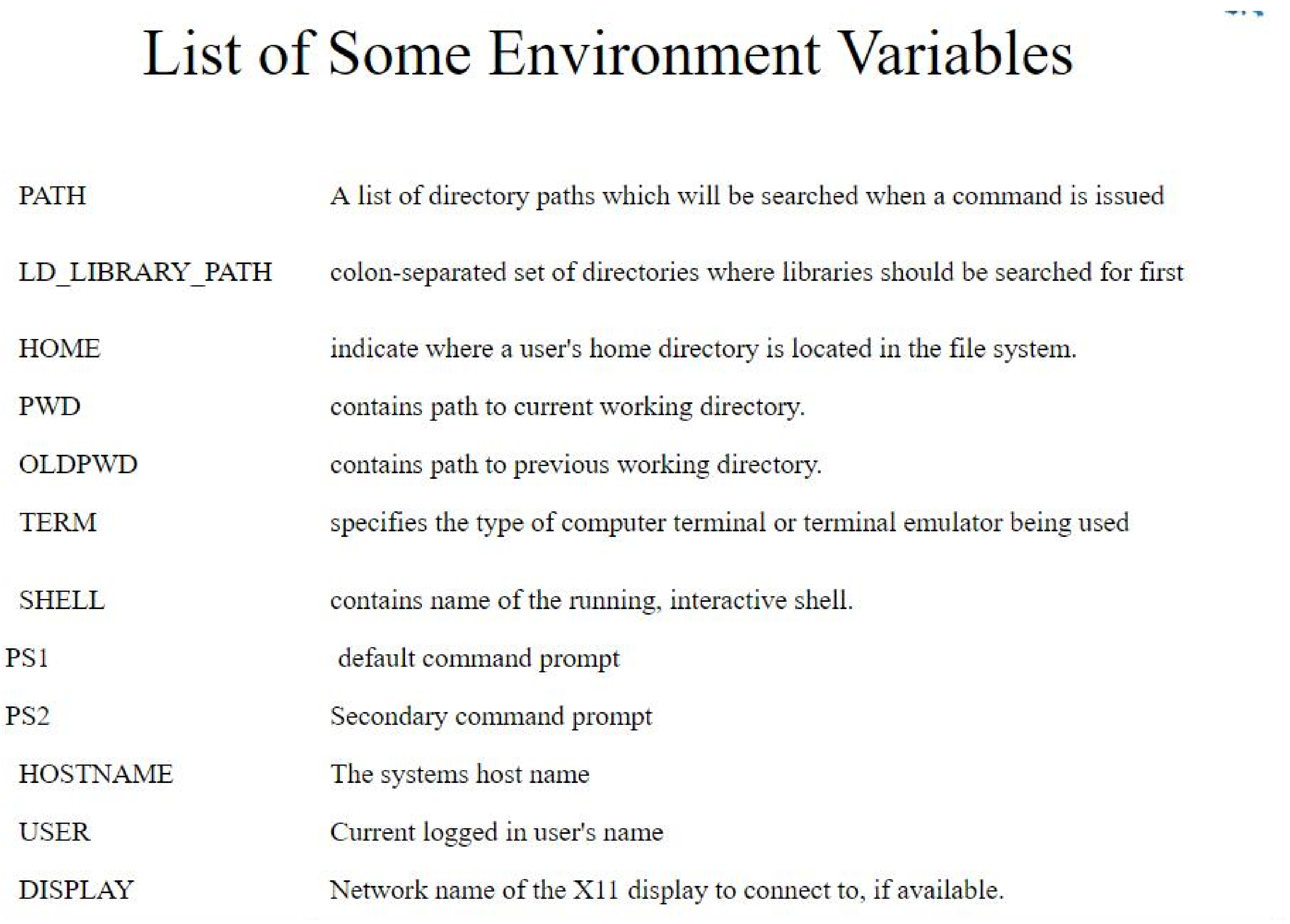
```
#!/bin/bash

echo $user_name
```

04 Basic shell scripting concepts



roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 85x19

```
roboticscorner@linux:~/files$ nano bash.sh
roboticscorner@linux:~/files$ nano bash.sh
roboticscorner@linux:~/files$ touch call_global.sh
roboticscorner@linux:~/files$ sudo chmod +x call_global.sh
[sudo] password for roboticscorner:
roboticscorner@linux:~/files$ nano call_global.sh
roboticscorner@linux:~/files$ nano call_global.sh
roboticscorner@linux:~/files$ source bash.sh
Hello Robotics Corner World
roboticscorner@linux:~/files$ ./call_global.sh
Robotics Corner
roboticscorner@linux:~/files$
```



List of Some Environment Variables

PATH	A list of directory paths which will be searched when a command is issued
LD_LIBRARY_PATH	colon-separated set of directories where libraries should be searched for first
HOME	indicate where a user's home directory is located in the file system.
PWD	contains path to current working directory.
OLDPWD	contains path to previous working directory.
TERM	specifies the type of computer terminal or terminal emulator being used
SHELL	contains name of the running, interactive shell.
PS1	default command prompt
PS2	Secondary command prompt
HOSTNAME	The systems host name
USER	Current logged in user's name
DISPLAY	Network name of the X11 display to connect to, if available.

quotations

- Single quotation: Enclosing characters in single quotes (') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.
- Double quotation: Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of ‘$’, ‘`’, ‘\’

quotation

Str1=‘echo $USER’

Echo “$str1” → echo $USER

Str2=“echo $USER”

Echo “$str2” → echo YOUR_USERNAME

Str3=`echo $USER`

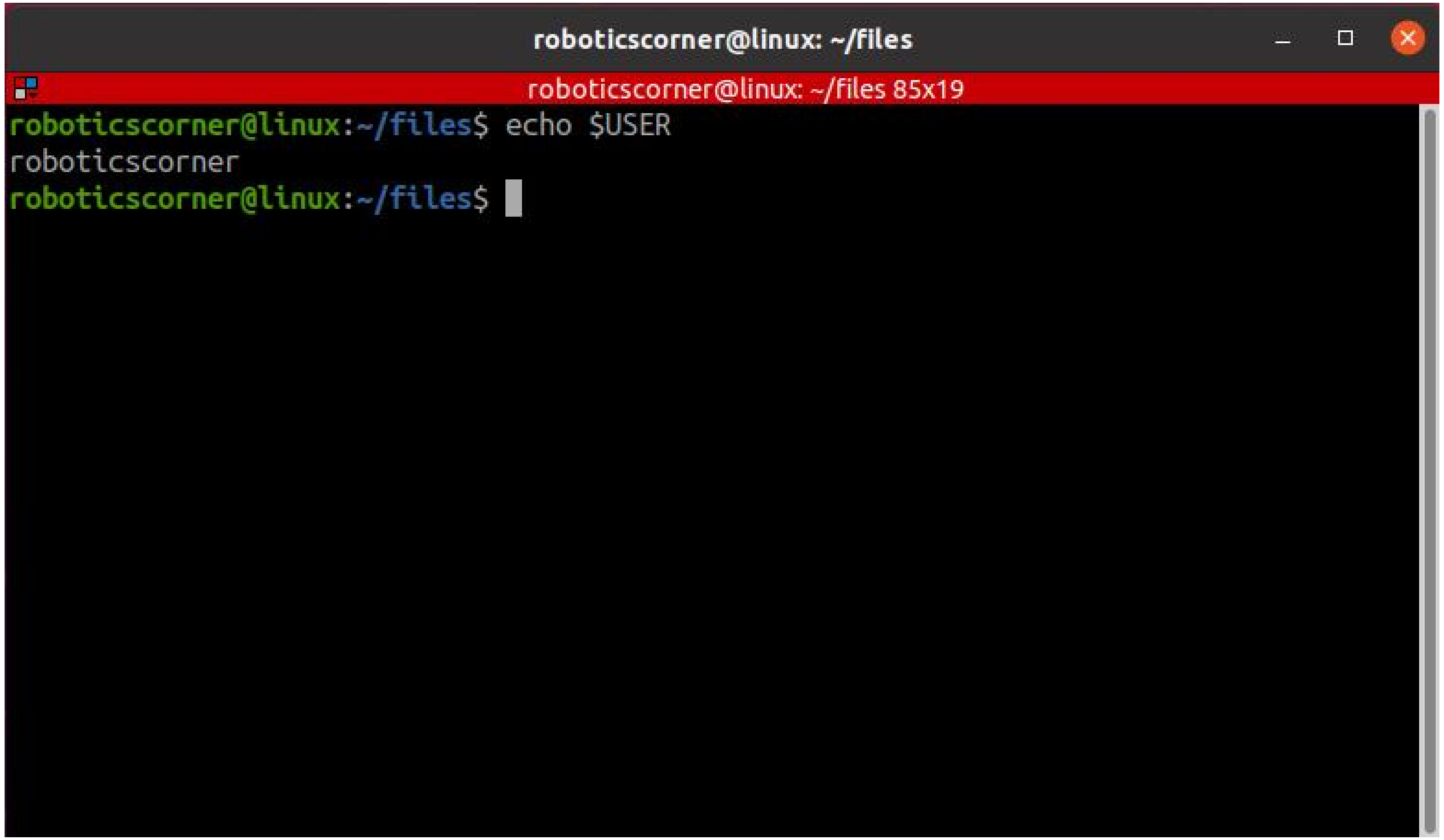
Echo $str3 → YOUR_USERNAME

Str4=$(echo $USER)

Echo “$str4” →YOUR_USERNAME

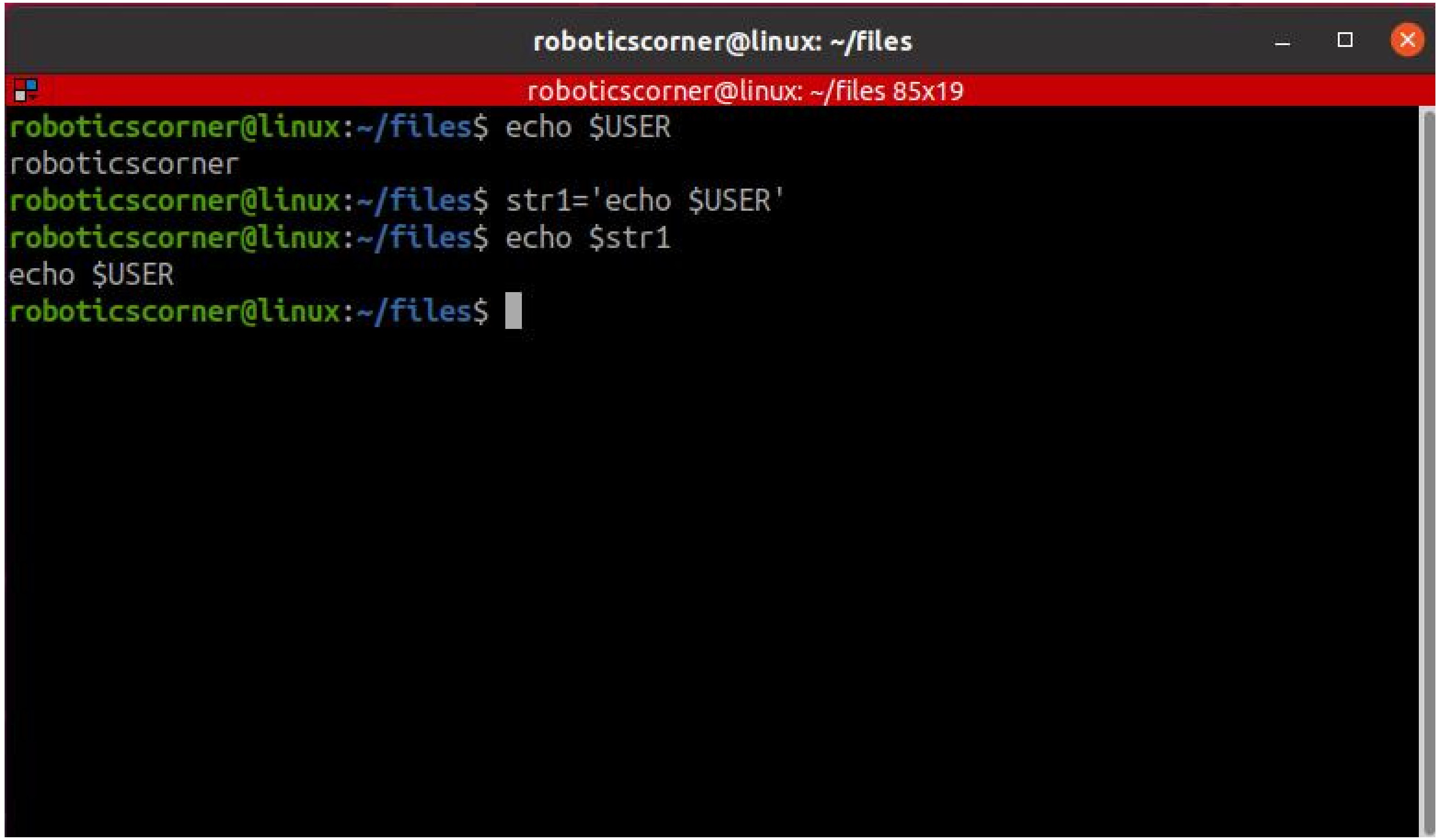
04

Basic shell scripting concepts


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 85x19

```
roboticscorner@linux:~/files$ echo $USER
roboticscorner
roboticscorner@linux:~/files$ 
```

04 Basic shell scripting concepts

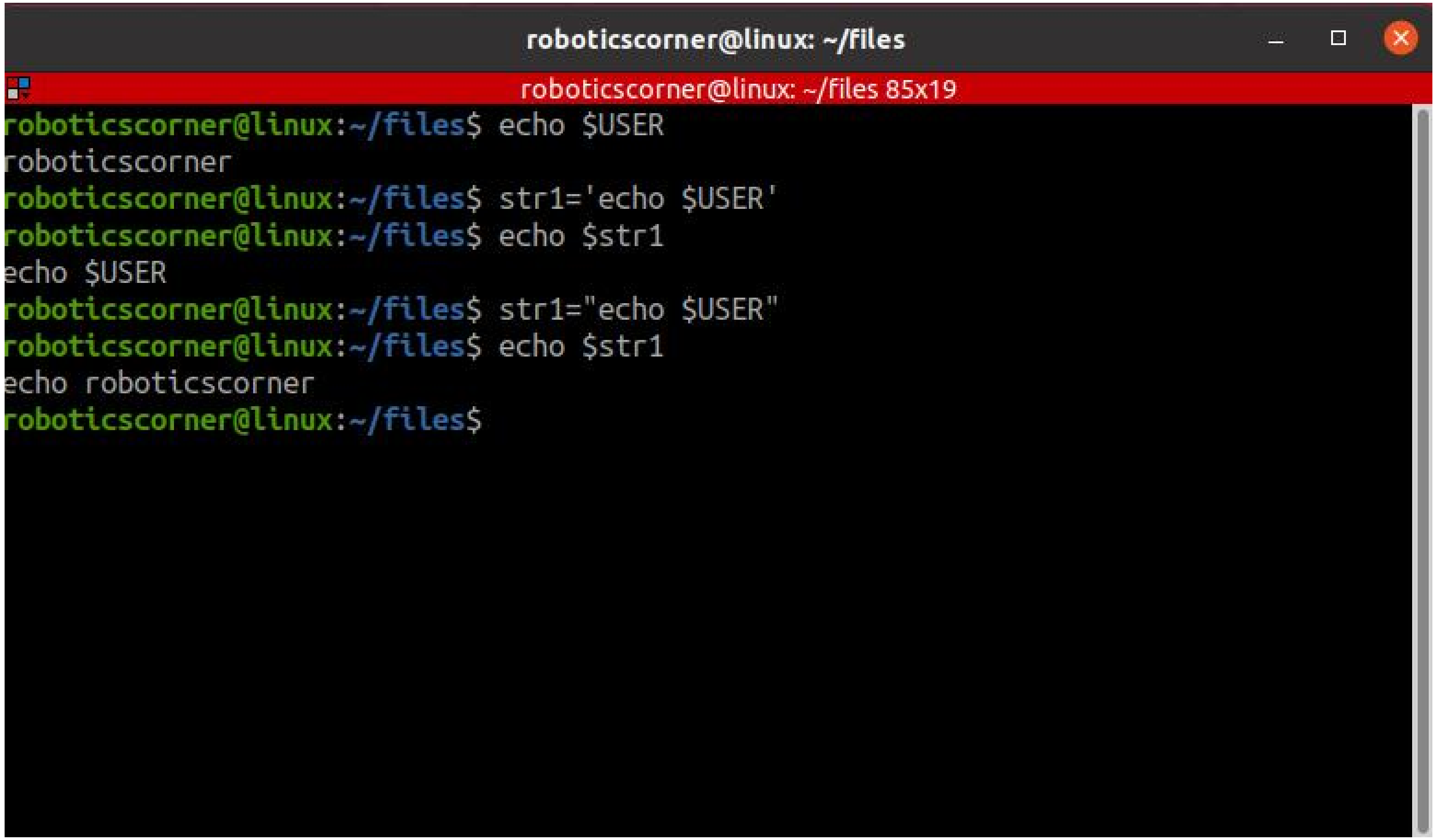


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 85x19

```
roboticscorner@linux:~/files$ echo $USER
roboticscorner
roboticscorner@linux:~/files$ str1='echo $USER'
roboticscorner@linux:~/files$ echo $str1
echo $USER
roboticscorner@linux:~/files$ 
```

04

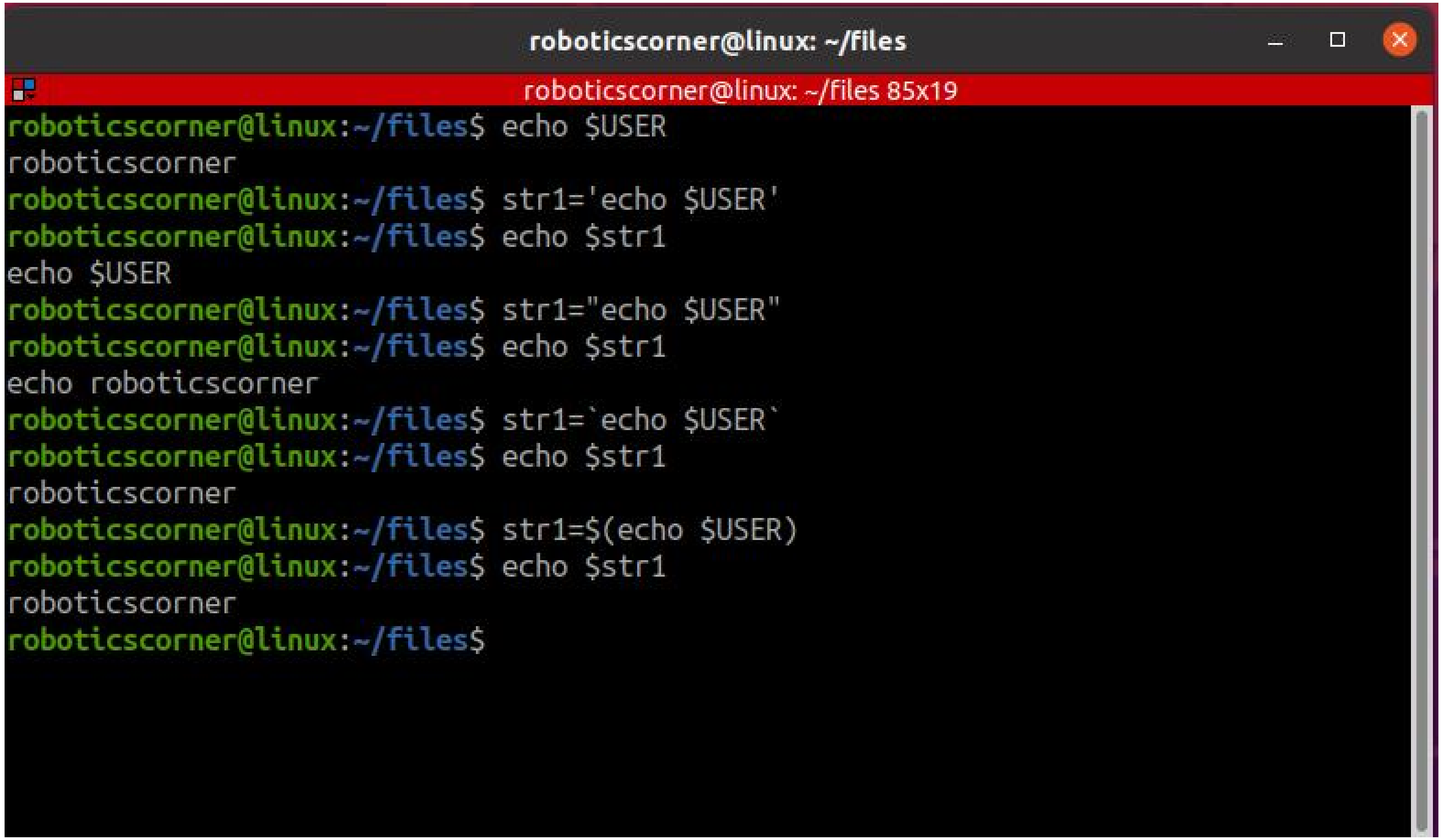
Basic shell scripting concepts



roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 85x19

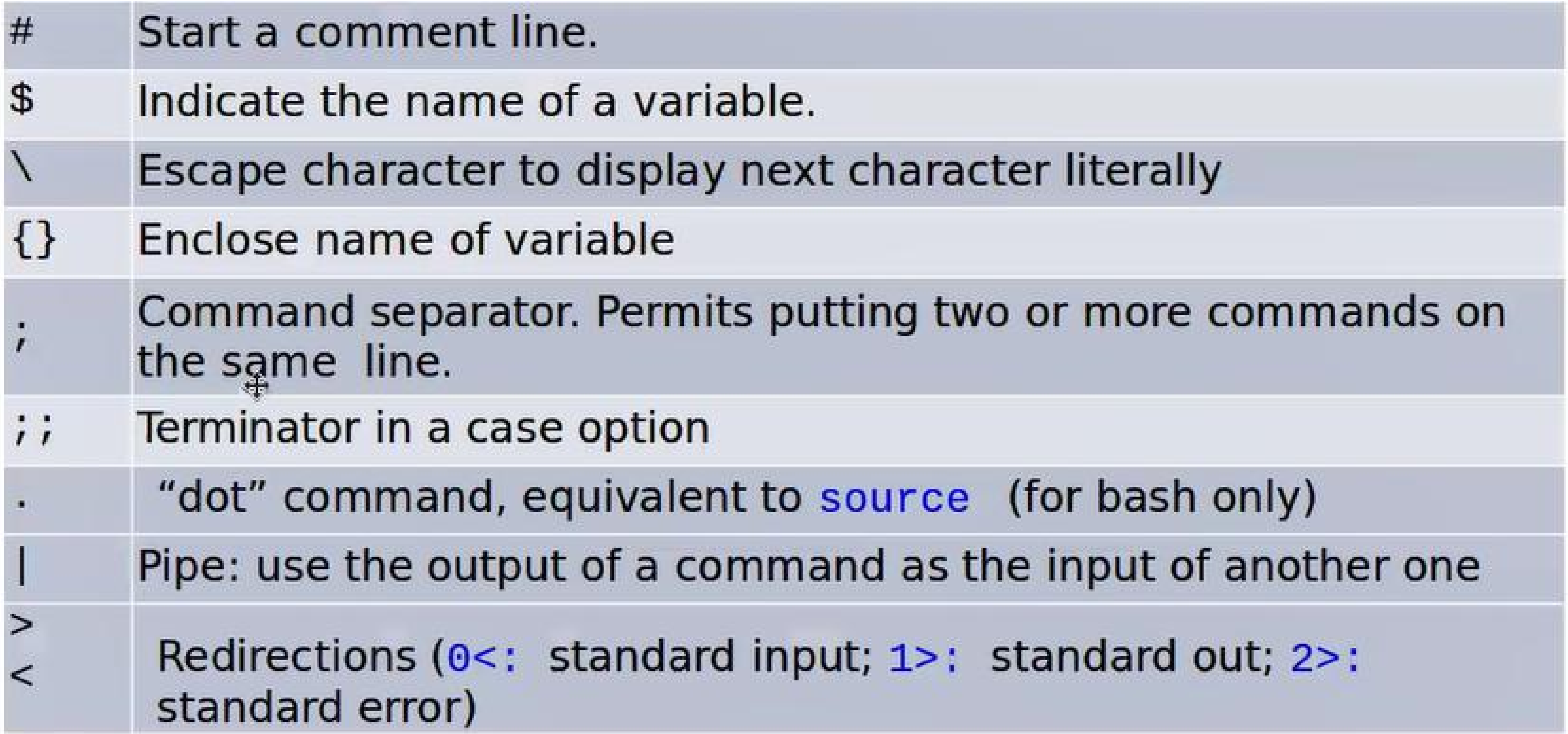
```
roboticscorner@linux:~/files$ echo $USER
roboticscorner
roboticscorner@linux:~/files$ str1='echo $USER'
roboticscorner@linux:~/files$ echo $str1
echo $USER
roboticscorner@linux:~/files$ str1="echo $USER"
roboticscorner@linux:~/files$ echo $str1
echo roboticscorner
roboticscorner@linux:~/files$
```

04 Basic shell scripting concepts


roboticscorner@linux: ~/files
roboticscorner@linux: ~/files 85x19

```
roboticscorner@linux:~/files$ echo $USER
roboticscorner
roboticscorner@linux:~/files$ str1='echo $USER'
roboticscorner@linux:~/files$ echo $str1
echo $USER
roboticscorner@linux:~/files$ str1="echo $USER"
roboticscorner@linux:~/files$ echo $str1
echo roboticscorner
roboticscorner@linux:~/files$ str1=`echo $USER`
roboticscorner@linux:~/files$ echo $str1
roboticscorner
roboticscorner@linux:~/files$ str1=$(echo $USER)
roboticscorner@linux:~/files$ echo $str1
roboticscorner
roboticscorner@linux:~/files$
```

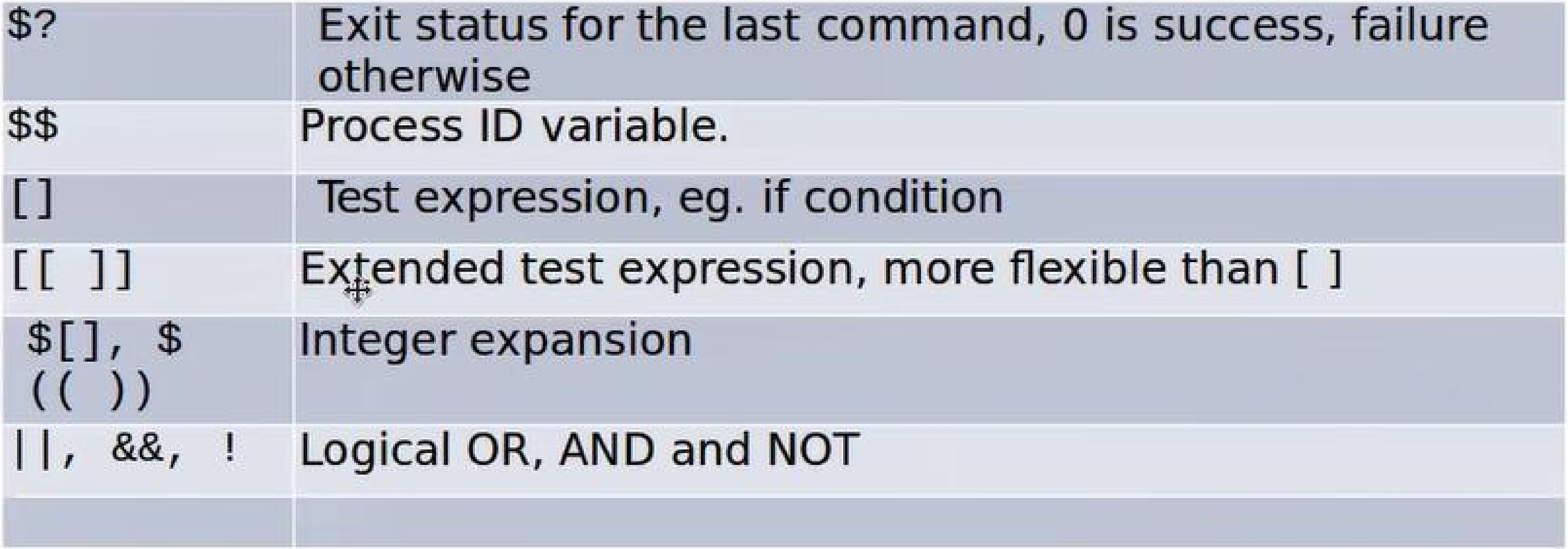
Special characters



#	Start a comment line.
$	Indicate the name of a variable.
\	Escape character to display next character literally
{}	Enclose name of variable
;	Command separator. Permits putting two or more commands on the same line.
;;	Terminator in a case option
.	"dot" command, equivalent to `source` (for bash only)
|	Pipe: use the output of a command as the input of another one
> <	Redirections (`0<:` standard input; `1>:` standard out; `2>:` standard error)

04

Basic shell scripting concepts



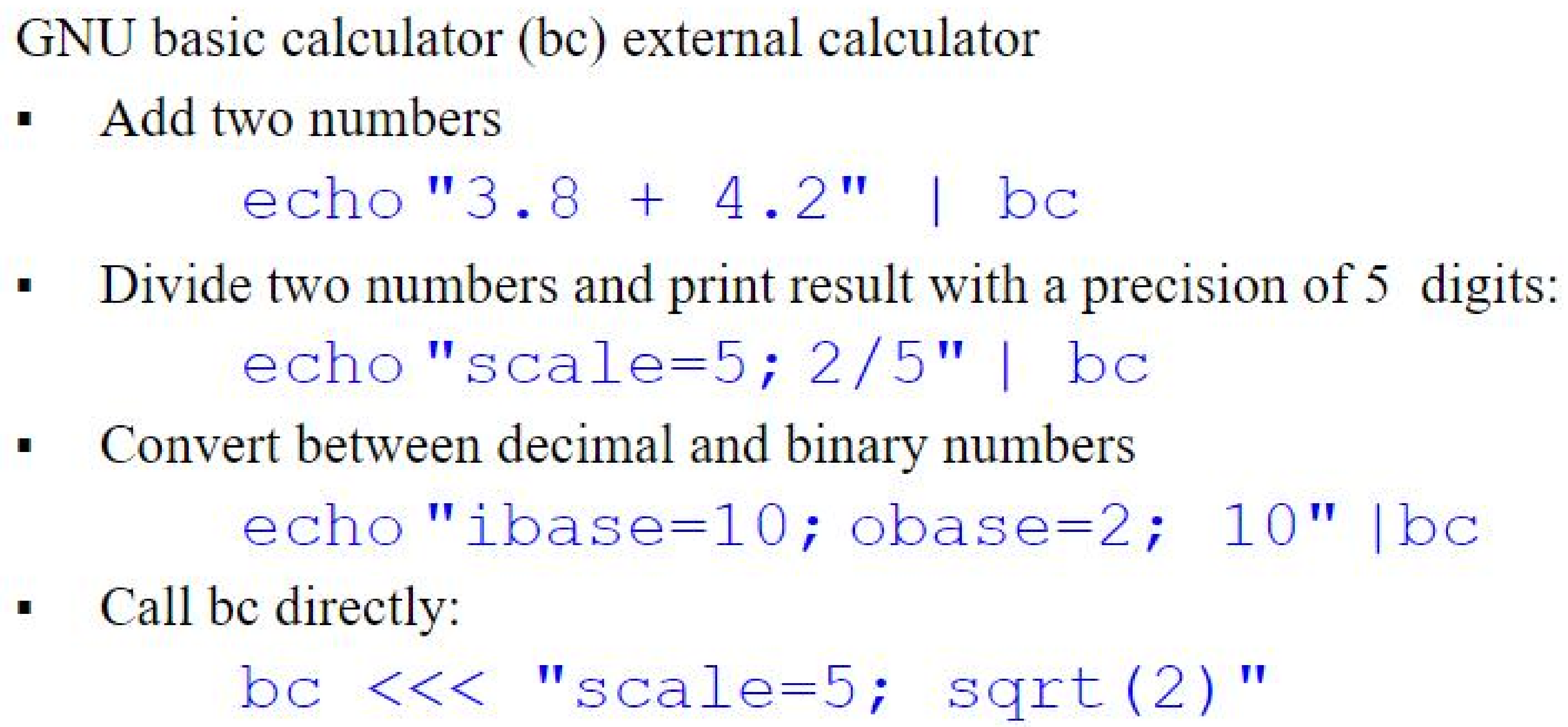
$?	Exit status for the last command, 0 is success, failure otherwise
$$	Process ID variable.
[]	Test expression, eg. if condition
[[ ]]	Extended test expression, more flexible than [ ]
$[], $ (( ))	Integer expansion
||, &&, !	Logical OR, AND and NOT

ROBOTICS CORNER

Integer Arithmetic Operations

- $((expression))
- $((n1+n2))
- $((n1/n2))
- $((n1-n2))
- Addition, Subtraction, Multiplication, Division, Exponentiation, Modulus

Floating-Point Arithmetic Operations



GNU basic calculator (bc) external calculator

- Add two numbers

```
echo "3.8 + 4.2" | bc
```

- Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

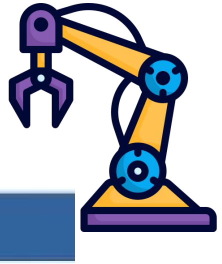
- Convert between decimal and binary numbers

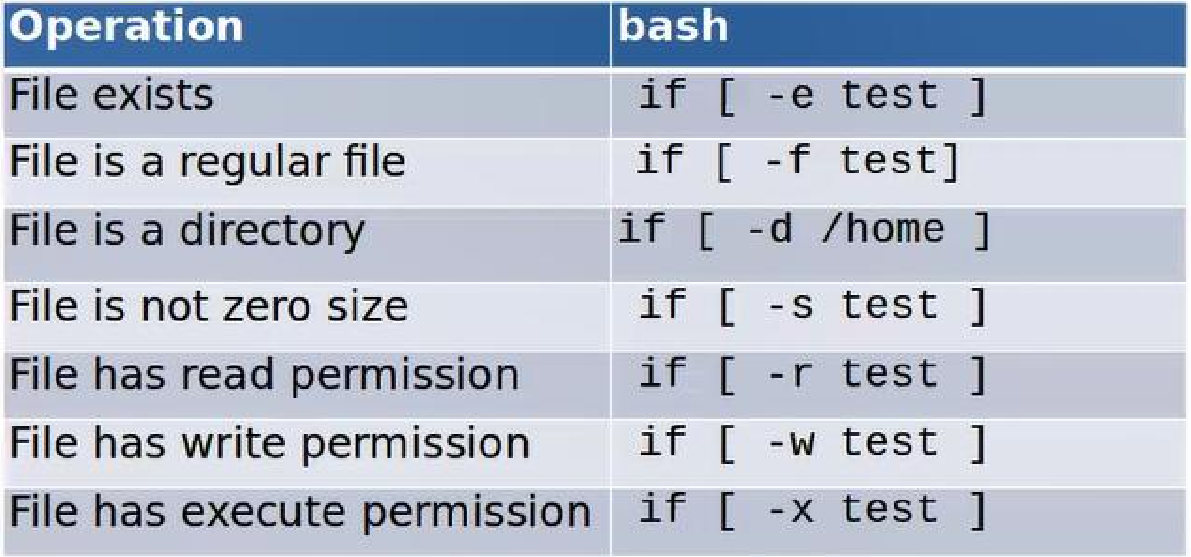
```
echo "ibase=10; obase=2; 10" |bc
```

- Call bc directly:

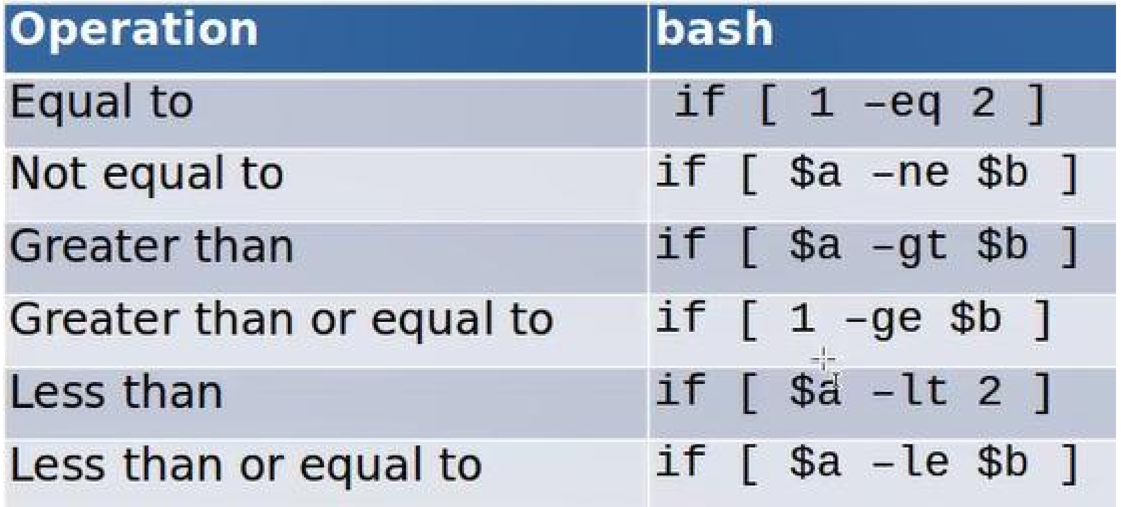
```
bc <<< "scale=5; sqrt(2)"
```

04 Basic shell scripting concepts

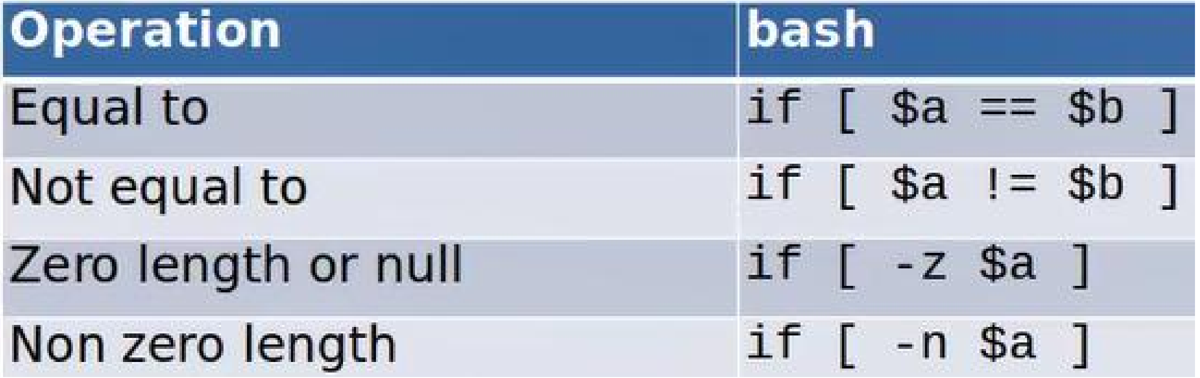




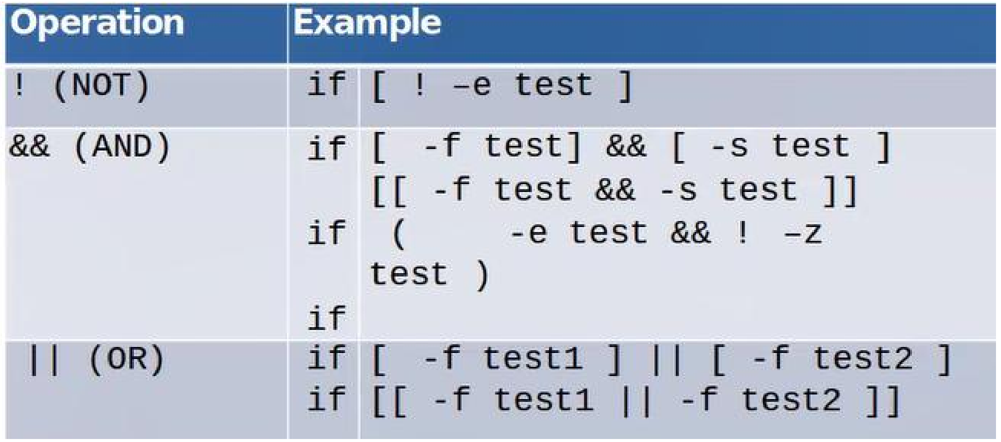
Operation	bash
File exists	if [-e test]
File is a regular file	if [-f test]
File is a directory	if [-d /home]
File is not zero size	if [-s test]
File has read permission	if [-r test]
File has write permission	if [-w test]
File has execute permission	if [-x test]



Operation	bash
Equal to	if [1 –eq 2]
Not equal to	if [$a –ne $b]
Greater than	if [$a –gt $b]
Greater than or equal to	if [1 –ge $b]
Less than	if [$a –lt 2]
Less than or equal to	if [$a –le $b]



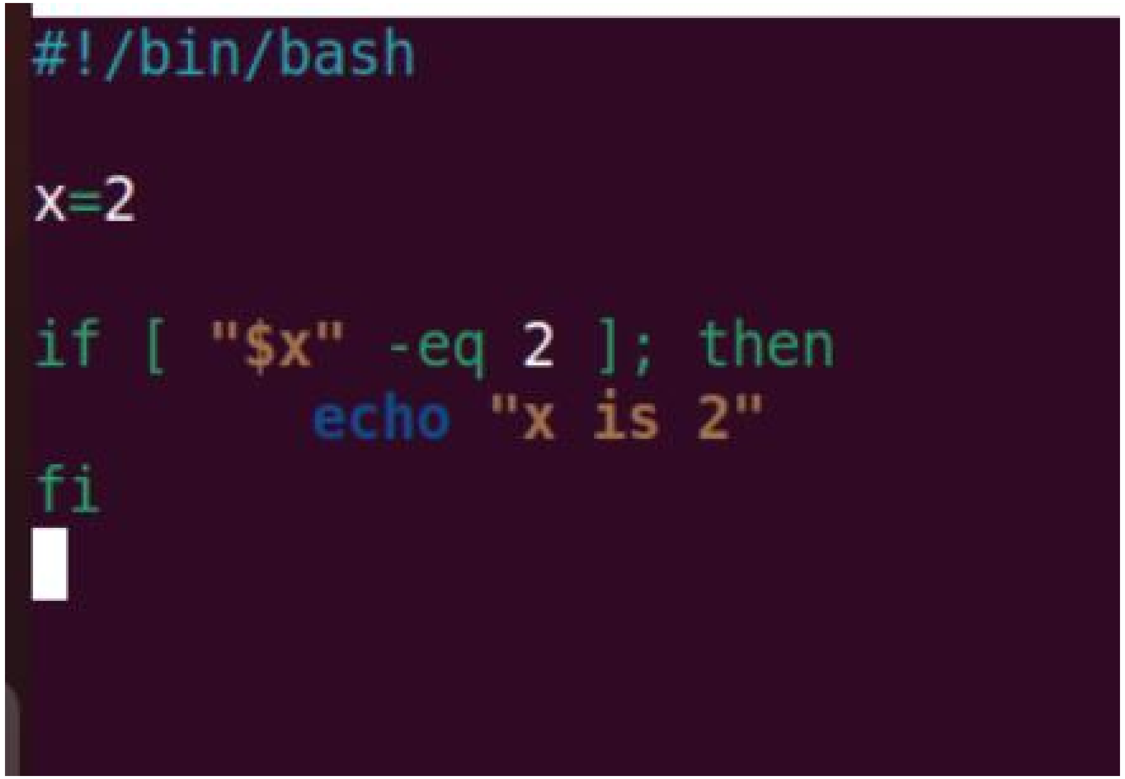
Operation	bash
Equal to	if [$a == $b]
Not equal to	if [$a != $b]
Zero length or null	if [-z $a]
Non zero length	if [-n $a]



Operation	Example	
! (NOT)	if	[! –e test]
&& (AND)	if if if	[-f test] && [-s test] [[-f test && -s test]] (-e test && ! –z test)
|| (OR)	if if	[-f test1] || [-f test2] [[-f test1 || -f test2]]

ROBOTICS CORNER

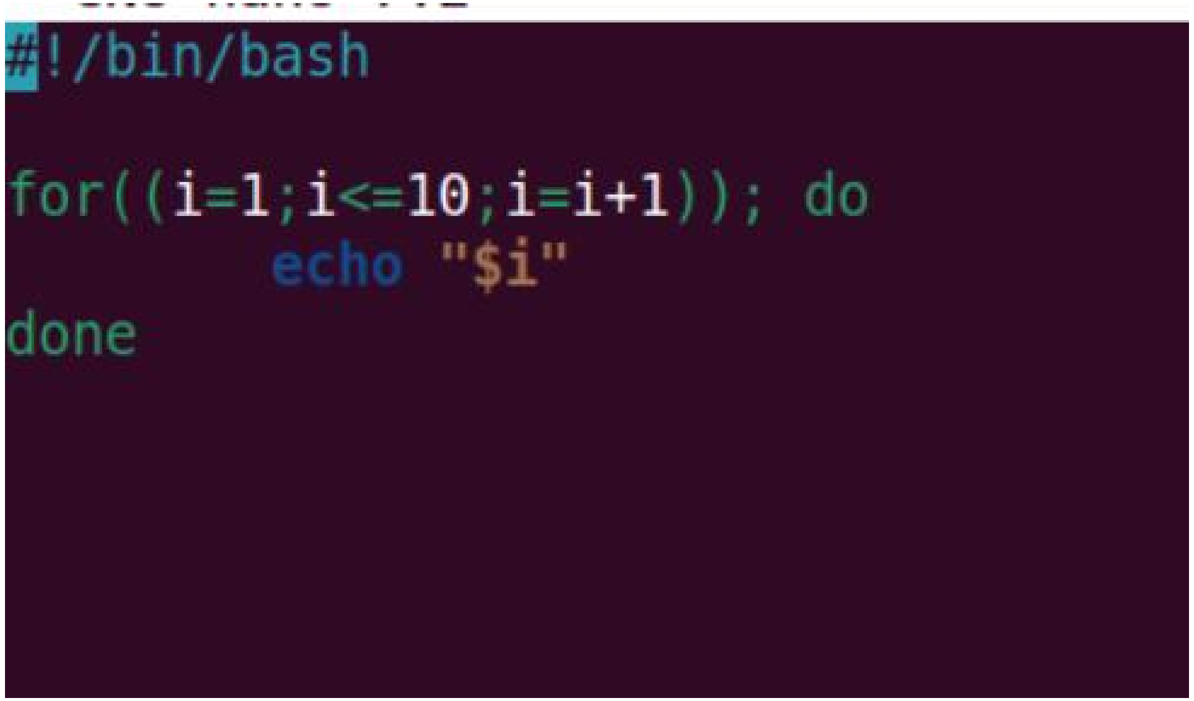
Conditional statements



```
#!/bin/bash

x=2

if [ "$x" -eq 2 ]; then
        echo "x is 2"
fi
```

Loops

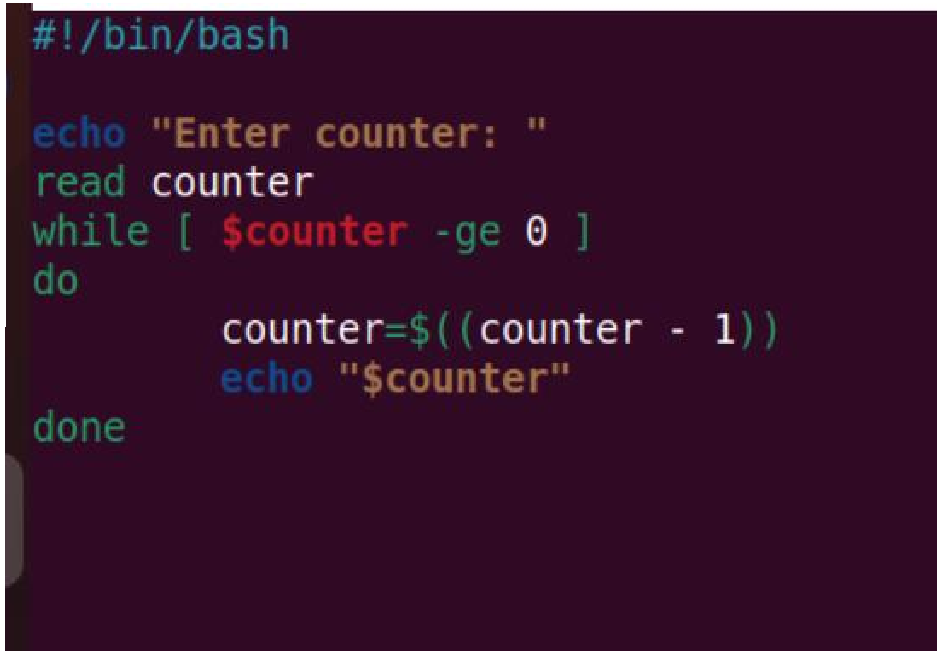
- A loop is a block of code that iterates a list of commands as long as the loop control condition stays true
- Loop constructs
- for, while and until



```
#!/bin/bash

for((i=1;i<=10;i=i+1)); do
        echo "$i"
done
```

While Loop

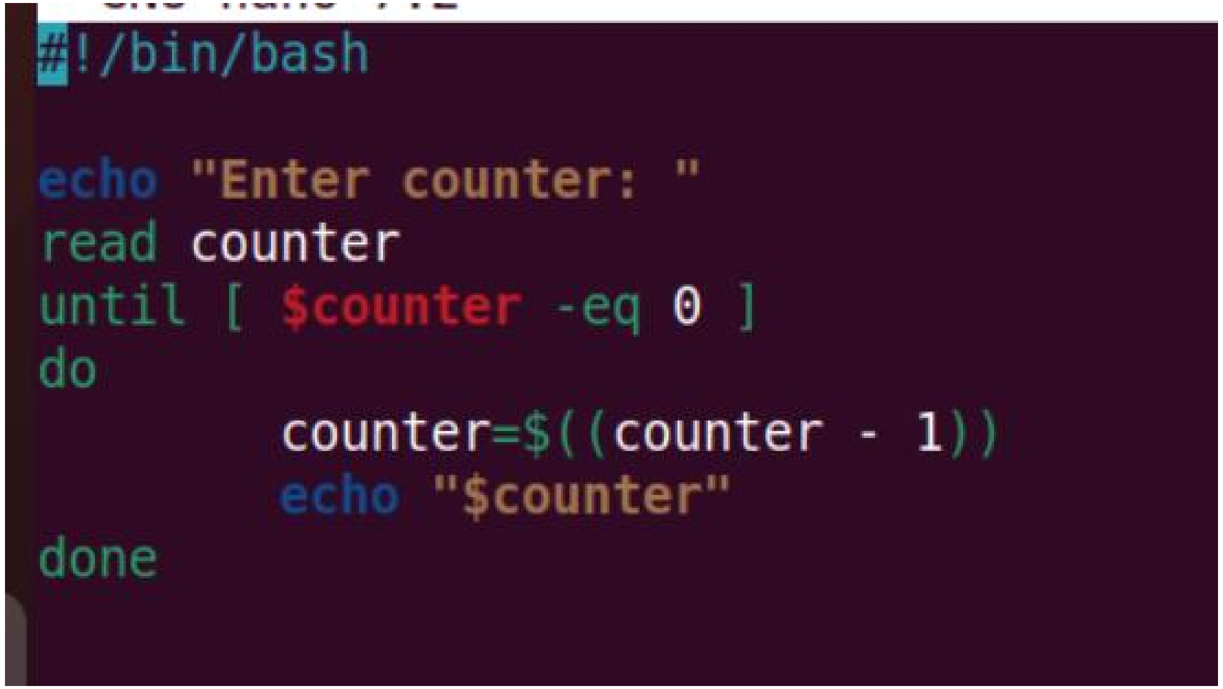
- The while construct test for a condition at the top of a loop and keeps going as long as that condition is
- true.
  In contrast to a for loop, a while is used when loop repetitions is not known beforehand.



```
#!/bin/bash

echo "Enter counter: "
read counter
while [ $counter -ge 0 ]
do
        counter=$((counter - 1))
        echo "$counter"
done
```

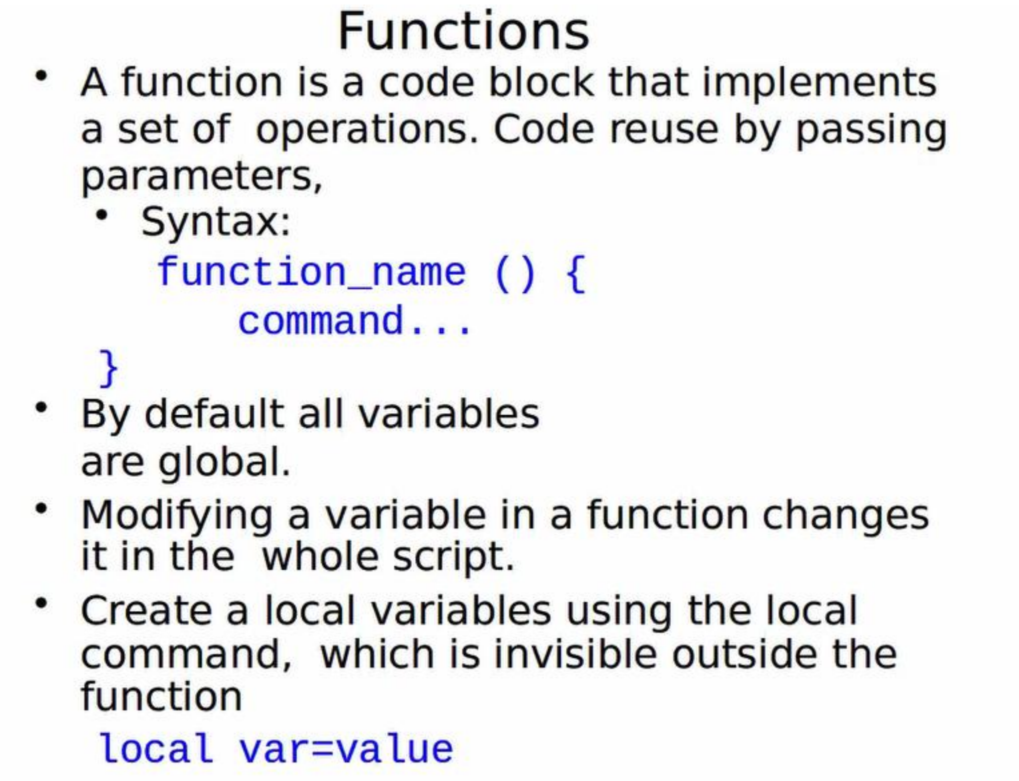
Until Loop

- The until construct test a condition at the top of a loop, and stops looping when the condition is met (opposite of while loop)



```
#!/bin/bash

echo "Enter counter: "
read counter
until [ $counter -eq 0 ]
do
        counter=$((counter - 1))
        echo "$counter"

done
```

04 Basic shell scripting concepts



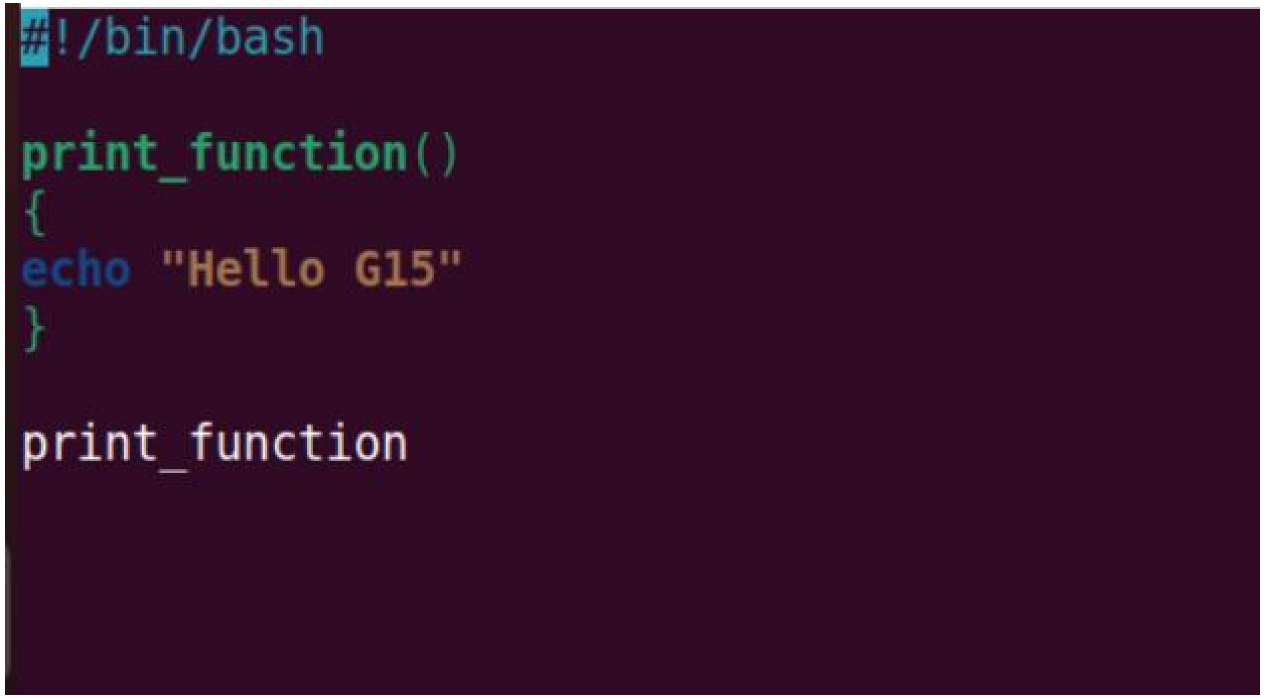
Functions

- A function is a code block that implements a set of operations. Code reuse by passing parameters,
  - Syntax:

```
function_name () {
      command...
}
```

- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- Create a local variables using the local command, which is invisible outside the function

```
local var=value
```



```
#!/bin/bash

print_function()
{
echo "Hello G15"
}

print_function
```

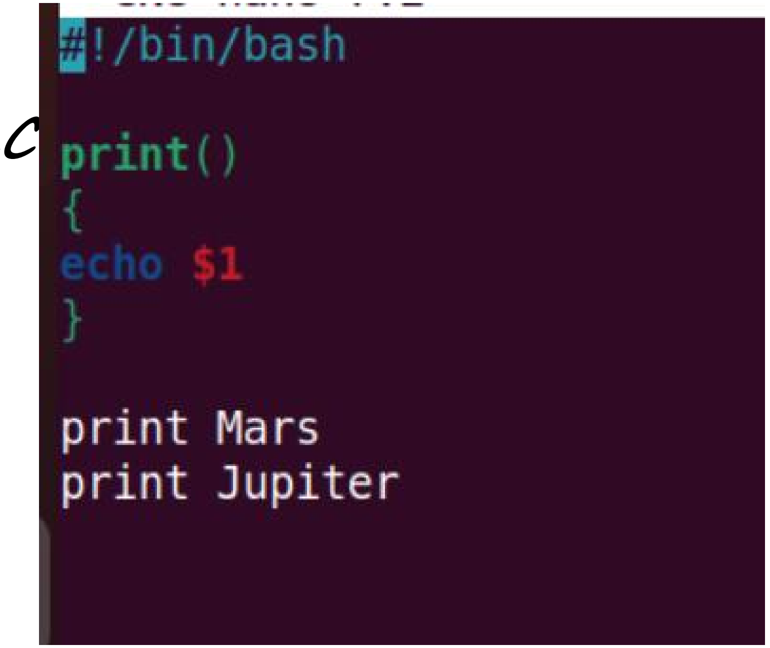
functions

```
#!/bin/bash
# Basic function
print_something () {
echo Hello I am a function
}
print_something
print_something
```

Passing Arguments

- Within the function they are accessible as $1, $2, etc.

```
#!/bin/bash
# Passing arguments to a func
print_something () {
echo Hello $1
}
print_something Mars
print_something Jupiter
```

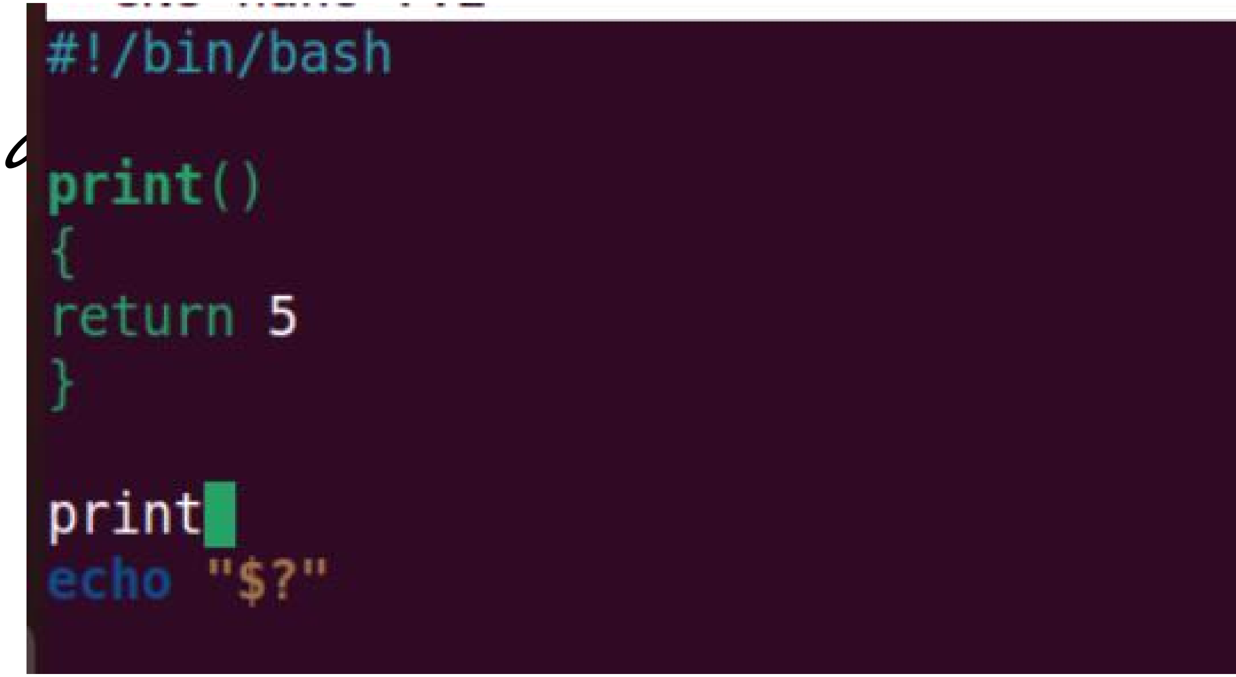


```
#!/bin/bash

print()
{
echo $1
}

print Mars
print Jupiter
```

ROBOTICS CORNER

Return Values

```
#!/bin/bash
# Setting a return status for a
print_something () {
echo Hello $1
return 5
}
print_something Mars
print_something Jupiter
echo The previous function has a return value of $?
```

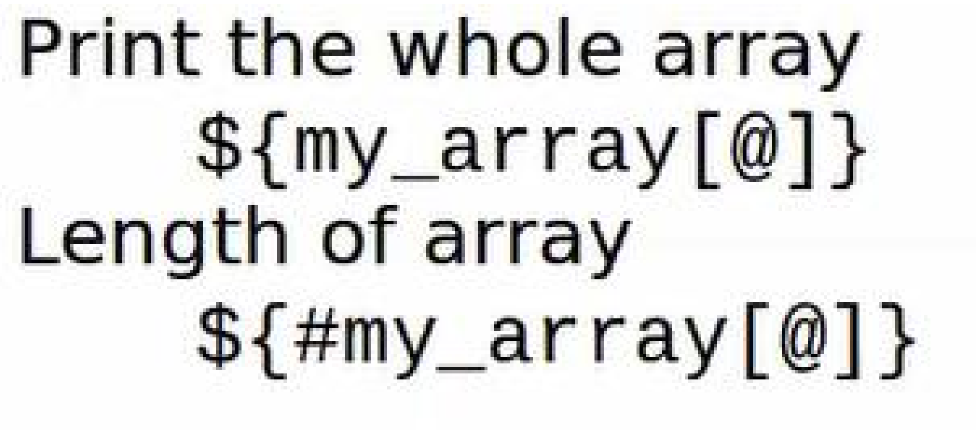


```
#!/bin/bash

print()
{
return 5
}

print
echo "$?"
```

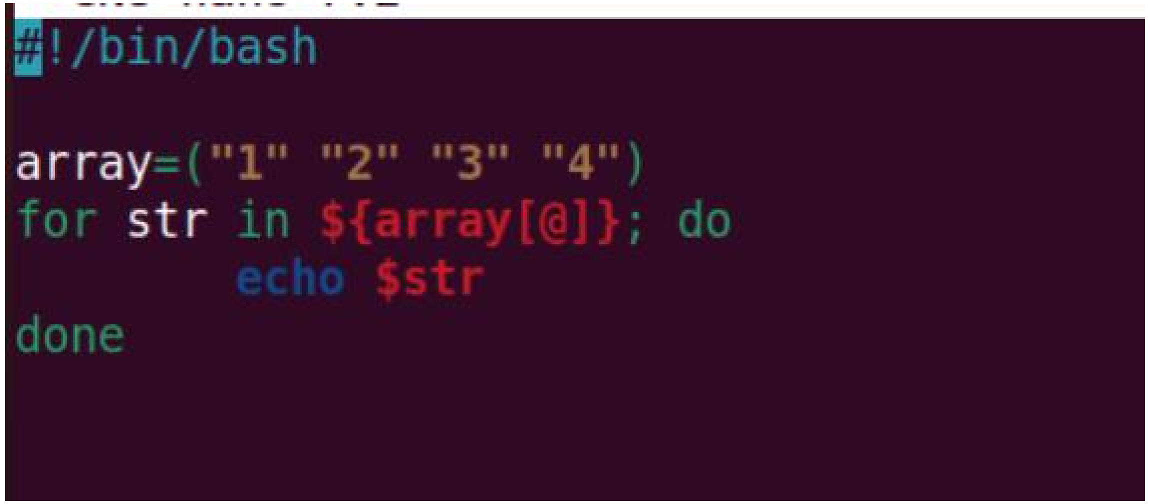
arrays

- myArray=("cat" "dog" "mouse" "frog")

```
for str in ${myArray[@]}; do
  echo $str
done
```



Print the whole array
    ${my_array[@]}
Length of array
    ${#my_array[@]}



```
#!/bin/bash

array=("1" "2" "3" "4")
for str in ${array[@]}; do
        echo $str
done
```

TASK1

- Implement a function that takes 2 inputs and performs
- 1. addition of the 2 numbers
- 2. Print the result
- 3. loops over the resulting addition and decrementing the value printing the result inside the loop

ROBOTICS CORNER

Assignment

- Create a directory called dip that has 5 files: f1 f2 f3 f4 f5
- Implement a bash script that lists all the files in this directory and deletes the directory with its content using a LOOP

ROBOTICS CORNER

POSIX

- A POSIX (Portable Operating System Interface) system refers to an operating system that adheres to a set of standards specified by the IEEE (Institute of Electrical and Electronics Engineers) in the POSIX family of standards. The POSIX standards define a set of APIs (Application Programming Interfaces) and other conventions for ensuring compatibility between different Unix-like operating systems.
- Shell and Utilities, File System Structure, Process Control, User and Group IDs, IO Operations, Networking, threads, and system adminstration

Thank You

Do you have any questions?

ROBOTICS CORNER

ROBOTICS CORNER

01211626904

Robotics Corner

Robotics Corner

www.roboticscorner.tech

Robotics Corner

Robotics Corner