Birla Institute of Technology & Science, Pilani
Hyderabad Campus

Intro & Shell Scripting

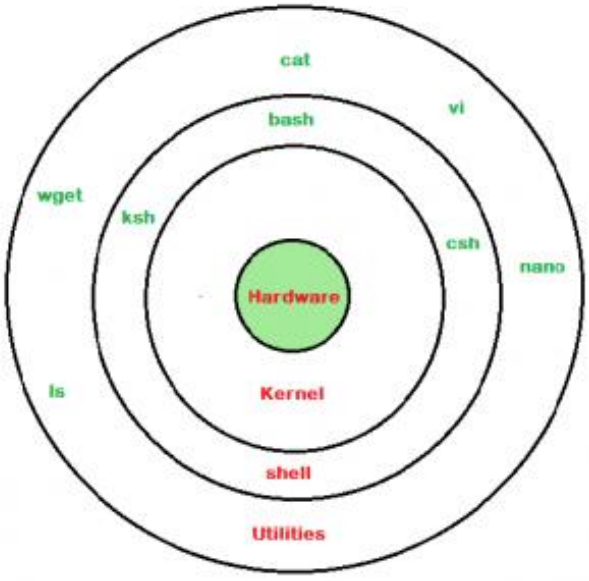
Linux shells

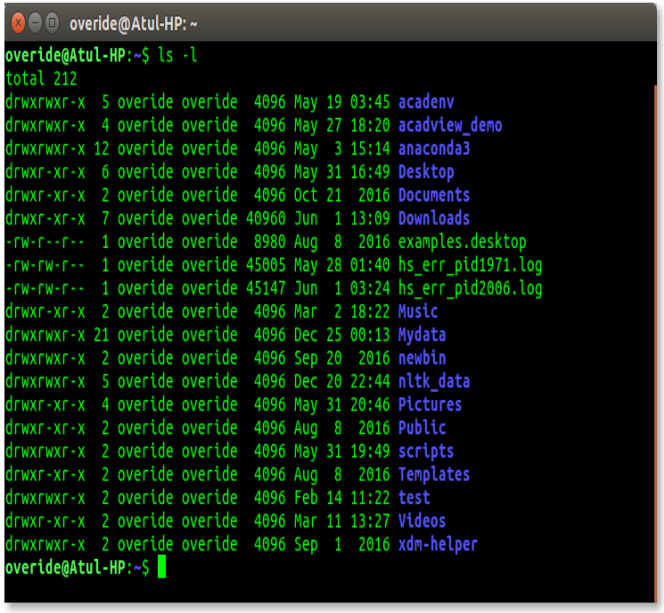
- Kernel
- Shell
- Terminal

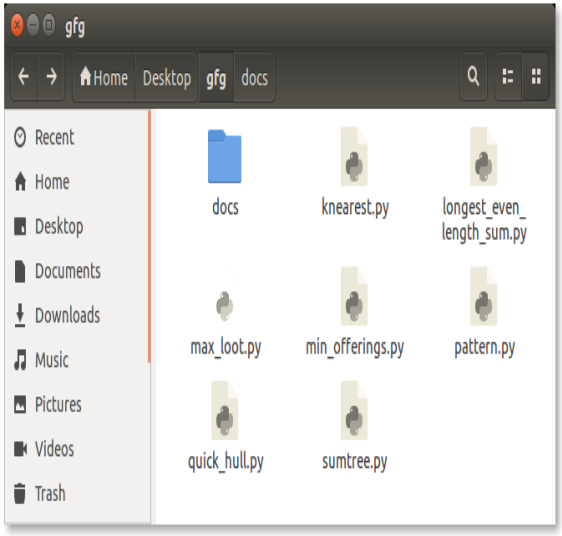
Complete Linux system = Kernel + GNU system utilities and libraries + other management scripts + installation scripts

- A shell is a user program that provides an interface for the user to use operating system services
- It is a Command Line Interpreter that executes commands read from input devices such as keyboards or from files

- Command Line Shell
- Graphical shell







Types of shells in Linux

- BASH (Bourne Again SHell) – It is the most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
- CSH (C SHell) – The C shell's syntax and its usage are very similar to the C programming language.
- KSH (Korn SHell) – The Korn Shell was also the base for the POSIX Shell standard specifications etc.

Shell scripting

innovate achieve lead

- Shells are interactive (they accept commands as input from users and execute them)
- To execute a bunch of commands routinely, we can write these commands in a file and can execute them in shell, which are called shell scripts or shell programs (saved as .sh files)

A shell script comprises the following elements:

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc

Basic shell commands in Linux

innovate achieve lead

- cat => concatenate
- more => filter (screenful)
- less => similar to more (backward and forward movement)
- head => Used to print the first N lines of a file. It accepts N as input and the default value of N is 10
- Tail => Used to print the last N-1 lines of a file. It accepts N as input and the default value of N is 10
- mkdir; cp; mv; rm; touch (create or update a file);
- grep (to search for a specific text in a file); sort; wc; cut (cut a specific part of a file)

Basic terminal navigation commands

innovate achieve lead

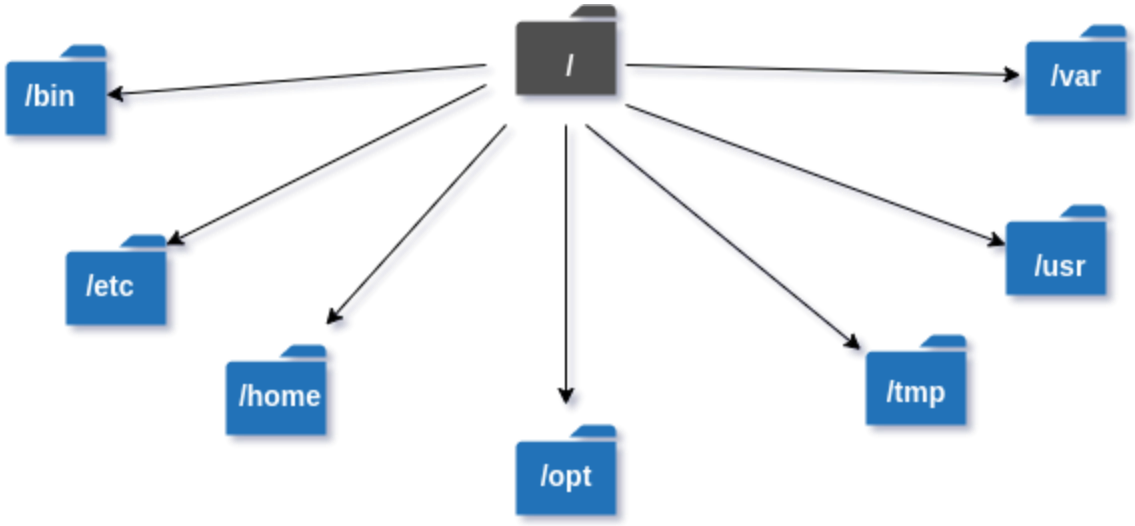
- **ls** : To get the list of all the files or folders.
- **ls -l:** Optional flags are added to **ls** to modify default behavior, listing contents in extended form **-l** is used for **"long" output**
- **ls -a:** Lists of all files including the hidden files, add **-a flag**
- **cd**: Used to change the directory.
- **du**: Show disk usage.
- **pwd**: Show the present working directory.
- **man**: Used to show the manual of any command present in Linux.
- **rmdir**: It is used to delete a directory if it is empty.
- **ln file1 file2**: Creates a physical link.
- **ln -s file1 file2**: Creates a symbolic link.
- **locate:** It is used to locate a file in Linux System
- **echo:** This command helps us move some data, usually text into a file.
- **df:** It is used to see the available disk space in each of the partitions in your system.
- **tar:** Used to work with tarballs (or files compressed in a tarball archive)

File permission commands

- **chown** : Used to change the owner of the file
- **chgrp** : Used to change the group owner of the file
- **chmod** : Used to modify the access/permission of a user

Linux Directory structure

- General files
- Directory files
- Device files



Directories	Description
/bin	binary or executable programs.
/etc	system configuration files.
/home	home directory. It is the default current directory.
/opt	optional or third-party software.
/tmp	temporary space, typically cleared on reboot.
/usr	User related programs.
/var	log files.

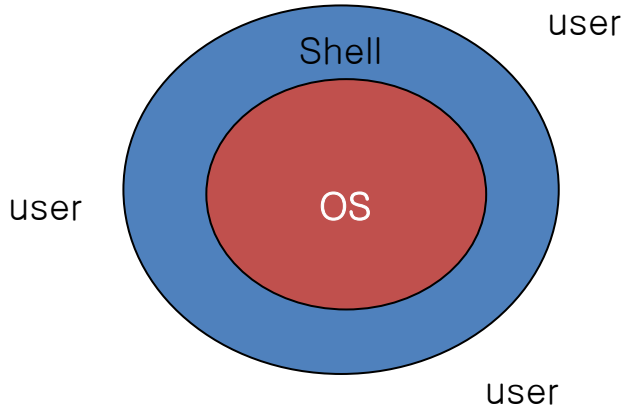
Directories	Description
/boot	It contains all the boot-related information files and folders such as conf, grub, etc.
/dev	It is the location of the device files such as dev/sda1, dev/sda2, etc.
/lib	It contains kernel modules and a shared library.
/lost+found	It is used to find recovered bits of corrupted files.
/media	It contains subdirectories where removal media devices are inserted.
/mnt	It contains temporary mount directories for mounting the file system.
/proc	It is a virtual and pseudo-file system to contains info about the running processes with a specific process ID or PID.
/run	It stores volatile runtime data.
/sbin	binary executable programs for an administrator.
/srv	It contains server-specific and server-related files.
/sys	It is a virtual file system for modern Linux distributions to store and allows modification of the devices connected to the system.

The “Shell” is simply *another program* on top of the kernel which provides a basic human-OS interface.

- It is a command interpreter
  - Built on top of the kernel
  - Enables users to run services provided by the UNIX OS
- In its simplest form, a series of commands in a file is a shell program that saves having to retype commands to perform common tasks.

How to know what shell you use

```
echo $SHELL
```



sh Bourne Shell (Original Shell) (*Steven Bourne of AT&T*)

bash Bourne Again Shell (*GNU Improved Bourne Shell*)

csh C-Shell (C-like Syntax)(*Bill Joy of Univ. of California*)

ksh Korn-Shell (Bourne+some C-shell)(*David Korn of AT&T*)

tcsh Turbo C-Shell (More User Friendly C-Shell).

To check shell:

- $ `echo $SHELL` (shell is a pre-defined variable)

To switch shell:

- $ exec shellname (e.g., `$ exec bash` or simply type `$ bash`)
- You can switch from one shell to another by just typing the name of the shell. `exit` return you back to previous shell.

sh (Bourne shell) was considered better for programming

csh (C-Shell) was considered better for interactive work.

tcsh and **korn** were improvements on c-shell and bourne shell respectively.

bash is largely compatible with sh and also has many of the nice features of the other shells

On many systems such as our LINUX clusters sh is symbolically linked to bash, /bin/sh -> /bin/bash

We recommend that you use sh/bash for writing new shell scripts but learn csh/tcsh to understand existing scripts.

Many, if not all, scientific applications require csh/tcsh environment (GUI, Graphics Utility Interface)

All Linux versions use the Bash shell (Bourne Again Shell) as the default shell

- **Bash/Bourn/ksh/sh prompt: $**

- **All UNIX system include C shell and its predecessor Bourne shell.**
  - **Csh/tcsh prompt: %**

`grep`

- Pattern searching
- Example: `grep 'boo' filename`

`sed`

- Text editing
- Example: `sed 's/XYZ/xyz/g' filename`

`awk`

- Pattern scanning and processing
- Example: `awk '{print $4, $7}' filename`

Hello world shell script

$ gedit helloworld.sh

```
#! /bin/sh // not mandatory
# The first example of a shell script
directory=`pwd`
echo Hello World!
echo The date today is `date`
echo The current directory is $directory
```

$ chmod +x hellowrold.sh

$ ./helloworld.sh

Quote characters in shell

There are three different quote characters with different behaviour. These are:

“ : double quote, weak quote. If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo “Today is:” `date``

Echo command

innovate achieve lead

- Echo command is used when trying to debug scripts.

Syntax : `echo {options} string`

Options: `-e` : expand \ (back-slash) special characters

`-n` : do not output a new-line at the end.

- String can be a “weakly quoted” or a ‘strongly quoted’ string. In the weakly quoted strings the references to variables are replaced by the value of those variables before the output.
- As well as the variables some special backslash_escaped symbols are expanded during the output. If such expansions are required the –e option must be used.

User Input in Shell Script Execution

```
echo "Please enter three filenames:”
read  filea fileb filec
echo “These files are used:$filea  $fileb
  $filec”
```

- Each read statement reads an entire line. In the above example if there are less than 3 items in the response the trailing variables will be set to blank ‘ ‘.
- Three items are separated by one space.

- The following script asks the user to enter his name and displays a personalised hello.

```
#!/bin/sh
echo "Who am I talking to?"
read user_name
echo "Hello $user_name"
```

- Try replacing " with ' in the last line to see what happens.

Programming features

- **Shell variables**
- **Operators**
- **Logic structures**:
  - **sequential logic** (for performing a series of commands)
  - **decision logic** (for branching from one point in a script to another)
  - **looping logic** (for repeating a command several times)
  - **case logic** (for choosing an action from several possible alternatives)

Variables

Three different types of variables

- Global Variables**: Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USERNAME, and PWD.**

  When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.

- Local Variables

  Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.

- Special Variables

  Reversed for OS, shell programming, etc. such as positional parameters $0, $1 …

Global variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing variable

Variable contents are accessed using '$':

e.g. $ echo $HOME

```
$ echo $SHELL
```

To see a list of your environment variables:

```
$ printenv
```

or:

```
$ printenv | more
```

Local variables

As in any other programming language, variables can be defined and used in shell scripts.

Unlike other programming languages, variables in Shell Scripts are not typed.

Examples :

```
a=1234  # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1 `  will perform arithmetic so b is 1235 now.
```

Note : +,-,/,*,**, % operators are available.

```
b=abcde # b is string
b='abcde' # same as above but much safer.
b=abc  def # will not work unless 'quoted'
b='abc def' # i.e. this will work.
```

IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND =

Referencing variables

- Having defined a variable, its contents can be referenced by the $ symbol. E.g. `${variable}` or simply `$variable`. When ambiguity exists $variable will not work. Use `${ }` the rigorous form to be on the safe side.
- Example:

```
a='abc'
b=${a}def # this would not have worked without the { }
          as it would try to access a variable named
          adef
```

Variable list , array

To create lists (array) – round bracket

$ set Y = (UNL 123 CS251)

To set a list element – square bracket

$ set Y[2] = HUSKER

To view a list element:

$ echo $Y[2]

Example:

```
#!/bin/sh
a=(1 2 3)
echo ${a[*]}
echo ${a[0]}
```

Results:

```
1 2 3
1
```

Positional parameters

- When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.
- $0 This variable that contains the name of the script
- $1, $2, ….. $n 1st, 2nd 3rd command line parameter
- $# Number of command line parameters
- $$ process ID of the shell
- $@ same as $* but as a list one at a time (see for loops later)
- $? Return code 'exit code' of the last command
- `Shift` command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

Example:

Invoke : `./myscript bits pilani hyderabad campus`

During the execution of `myscript` variables $1 $2 $3 $4 will contain the values `bits, pilani, Hyderabad, campus` respectively.

```
vi myinputs.sh
    #! /bin/sh
    echo Total number of inputs: $#
    echo First input: $1
    echo Second input: $2

chmod u+x myinputs.sh

myinputs.sh HUSKER UNL CSE
      Total number of inputs: 3
      First input: HUSKER
      Second input: UNL
```

Shell operators

- The Bash/Bourne/ksh shell operators are divided into three groups:
  - defining and evaluating operators
  - arithmetic operators
  - redirecting and piping operators

Defining and evaluating

- A shell variable take on the generalized form variable=value (except in the C shell).

```
$ set x=37; echo $x
37
$ unset x; echo $x
x: Undefined variable.
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ set mydir=`pwd`; echo $mydir
```

Pipes & Redirecting

- **Piping: An important early development in Unix , a way to pass the output of one tool as the input of another.**

```
$ who | wc -l
```

By combining these two tools, giving the wc command the output of who, you can build a new command to list the number of users currently on the system

- **Redirecting via angle brackets:** Redirecting input and output follows a similar principle to that of piping except that redirects work with files, not commands.

```
tr '[a-z]' '[A-Z]' < $in_file > $out_file
```

The command must come first, the *in_file* is directed in by the less_than sign (<) and the *out_file* is pointed at by the greater_than sign (>).

Arithmetic operators

expr supports the following operators:

- **arithmetic operators: +,-,*,/,%**
- **comparison operators: <, <=, ==, !=, >=, >**
- **boolean/logical operators: &, |**
- **parentheses: (,)**
- **precedence is the same as C, Java**

Ex:

vi math.sh

```
#!/bin/sh
count=5
count=`expr $count + 1 `
echo $count
```

chmod u+x math.sh

math.sh

Ex:

vi real.sh

```
#!/bin/sh
a=5.48
b=10.32
c=`echo "scale=2; $a + $b" |bc`
echo $c
```

chmod u+x real.sh

./real.sh

```
15.80
```

Arithmetic operations in shell scripts

var++ ,var-- , ++var , --var	post/pre increment/decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& , |	bitwise AND, OR
&& ||	logical AND, OR

Logic structures in shell scripts

- **Sequential logic:** to execute commands in the order in which they appear in the program
- **Decision logic:** to execute commands only if a certain condition is satisfied
- **Looping logic:** to repeat a series of commands for a given number of times
- **Case logic:** to replace “if then/else if/else” statements when making numerous comparisons

Conditional statements (if constructs)

```
if command executes successfully
then
        execute command
elif this command executes successfully
then
        execute this command
        and execute this command
else
        execute default command
fi
```

However- elif and/or else clause can be omitted.

SIMPLE EXAMPLE:

```
if date | grep “Fri”
then
        echo “It’s Friday!”
fi
```

FULL EXAMPLE:

```
if   [  “$1”  ==  “Monday”  ]
then
        echo “The typed argument is Monday.”
elif [ “$1” == “Tuesday” ]
 then
        echo “Typed argument is Tuesday”
 else
        echo “Typed argument is neither Monday nor Tuesday”
fi
```

Note: = or == will both work in the test but == is better for readability.

Tests

String and numeric comparisons used with test or [[ ]] which is an alias for test and also [ ] which is another acceptable syntax

string1 = string2	True if strings are identical
String1 == string2	…ditto….
string1 !=string2	True if strings are not identical
string	Return 0 exit status (=true) if string is not null
-n string	Return 0 exit status (=true) if string is not null
-z string	Return 0 exit status (=true) if string is null

- int1 -eq int2 — Test identity
- int1 -ne int2 — Test inequality
- int1 -lt int2 — Less than
- int1 -gt int2 — Greater than
- int1 -le int2 — Less than or equal
- int1 -ge int2 — Greater than or equal

tests with logical operators || (or) and && (and)

Syntax: if cond1 && cond2 || cond3 …

An alternative form is to use a compound statement using the –a and –o keywords, i.e.

if cond1 –a cond22 –o cond3 …

Where cond1,2,3 .. Are either commands returning a a value or test conditions of the form [ ] or test …

Examples:

```
if  date | grep “Fri”  &&  `date +’%H’` -gt 17
then
    echo “It’s Friday, it’s home time!!!”
fi

if [ “$a” –lt 0 –o “$a” –gt 100 ]     # note the spaces around ] and [
then
    echo “ limits exceeded”
fi
```

File enquiry operations

-d file	Test if file is a directory
-f file	Test if file is not a directory
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists
-z file	Test if the file has zero length

All these conditions return true if satisfied and false otherwise.

Decision Logic

```
#!/bin/sh
   if [ "$#" -ne 2 ] then
          echo $0 needs two parameters!
          echo You are inputting $# parameters.
    else
          par1=$1
          par2=$2
    fi
    echo $par1
    echo $par2
```

```
#! /bin/sh
     #  number is positive, zero or negative
     echo –e "enter a number:\c"
     read number
     if [ “$number” -lt 0 ]
     then
          echo "negative"
     elif [ “$number” -eq 0 ]
     then
          echo zero
     else
            echo positive
     fi
```

Loops

- Loop is a block of code that is repeated a number of times.
- The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (for loops) or until a particular condition is satisfied (while and until loops)
- To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

for loop

Syntax:

```
for  arg in list
do
    command(s)
        ...
done
```

Where the value of the variable ***arg*** is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.

Example:

```
for i in 3 2 5 7
do
    echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

while loop

- A different pattern for looping is created using the while statement
- The while statement best illustrates how to set up a loop to test repeatedly for a matching condition
- The while loop tests an expression in a manner similar to the if statement
- As long as the statement inside the brackets is true, the statements inside the do and done statements repeat

Syntax:

```
while this_command_execute_successfully
 do
        this command
        and this command
 done
```

EXAMPLE:

```
while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
do
        i=`expr $i - 1`
done
```

- **Example:**

```
#!/bin/sh
for person in Bob Susan Joe Gerry
do
   echo Hello $person
done
```

Output:

```
Hello Bob
Hello Susan
Hello Joe
Hello Gerry
```

- **Adding integers from 1 to 10**

```
#!/bin/sh
i=1
sum=0
while [ "$i" -le 10 ]
  do
  echo Adding $i into the sum.
  sum=`expr $sum + $i `
          i=`expr $i + 1 `
  done
echo The sum is $sum.
```

until loops

- The syntax and usage is almost identical to the while-loops.
- Except that the block is executed until the test condition is satisfied, which is the opposite of the effect of test condition in while loops.
- Note: You can think of *until* as equivalent to *not_while*

Syntax:

```
until test
    do
      commands ….
    done
```

Switch/Case Logic

- The switch logic structure simplifies the selection of a match when you have a list of choices
- It allows your program to perform one of many actions, depending upon the value of a variable

Case statements

The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
case argument in
    pattern 1) execute this command
               and this
               and this;;
    pattern 2) execute this command
               and this
               and this;;
    esac
```

Functions

```
#!/bin/sh
  sum() {
    x=`expr $1 + $2`
    echo $x
    }

sum 5 3
echo "The sum of 4 and 7 is `sum 4 7`"
```

Functions are a way of grouping together commands so that they can later be executed via a single reference to their name. If the same set of instructions have to be repeated in more than one part of the code, this will save a lot of coding and also reduce possibility of typing errors.

SYNTAX:

```
functionname()
{
    block of commands
}
```