# **What is Shell?**

A shell is special user program which provide an interface to user to use operating system services. Shell accept human readable commands from user and convert them into something which kernel can understand. It is a **command language interpreter** that execute commands read from input devices such as keyboards or from files. The shell gets started when the user logs in or start the terminal.

## 

***Architecture of Linux***

## Shell is broadly classified into two categories –

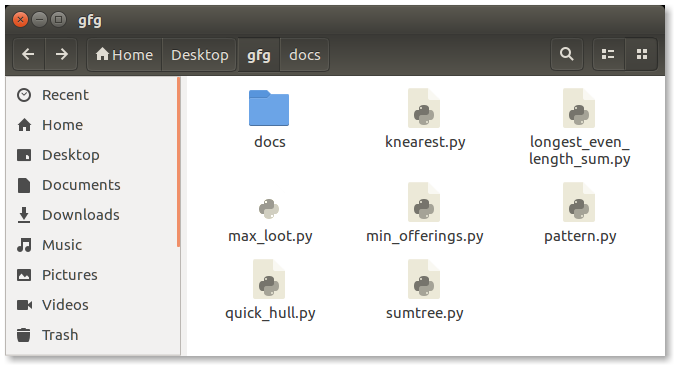
* + Graphical shell
  + Command Line Shell

**Graphical Shells**

One exciting aspect of Linux unlike with Windows and Mac OS, is its support for numerous number of desktop environments. A Desktop Environment is an implementation of the desktop metaphor built as a collection of different user and system programs running on top of an operating system, and share a common **GUI (Graphical User Interface)**, also known as a **graphical shell**

**Graphical shells** provide means for manipulating programs based on **graphical user interface (GUI)**, by allowing for operations such as opening, closing, moving and resizing windows, as well as switching focus between windows. Window OS or Ubuntu OS can be considered as good example which provide GUI to user for interacting with program. User do not need to type in command for every actions.

A typical GUI in Ubuntu system –



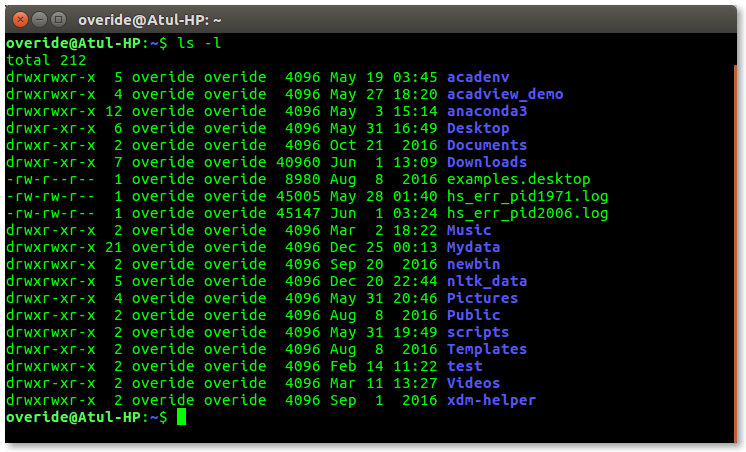
There are several different Desktop Environment available for Linux. The default Desktop Environment for Ubuntu is GNOME.

To check Desktop Environment of your system. Open the terminal and copy paste this command:

**echo $XDG\_CURRENT\_DESKTOP**

**Command Line Shell**

There are several operating systems available like **Windows, Linux,** etc. Each operating system in turn has a command processor which executes its commands. Thus, when a user types a command, the command processor which is the part of the operating system that accepts it. It will verify the validity of the command and will execute it if it is a valid command or gives an error warning if it is not. **Windows** operating systems have **command.com** as default command-line interpreter in the operating systems while **Unix** and **Linux** operating systems have the **C shell (CSH)**, the **Bourne shell**, and the **Bourne Again shell (BASH)**. A special program called **Terminal** in linux/macOS or **Command Prompt** in Windows OS for accessing a command line interface by a user is provided to type in the human readable commands such as “cat”, “ls” etc. and then it is being execute. The result is then displayed on the terminal to the user. A terminal in Ubuntu 16.4 system looks like this –



In above screenshot “ls” command with “-l” option is executed.

It will list all the files in current working directory in long listing format.

# **Shell Scripting**

A shell script is a computer program designed to be run by the Unix/Linux.

Whenever you find yourself doing the same task over and over again you should use shell scripting.

There are different types of shells available for linux. These are:

* The Bourne Shell
* The C Shell
* The Korn Shell
* Bourne-Again Shell

Evolution of Shells

**1977**

The Bourne shell was introduced. The **Bourne shell(sh)**, by Stephen Bourne at AT&T Bell Labs for V7 UNIX, remains a useful shell today (in some cases, as the default root shell).

* Denoted as **sh**
* Non-root user default prompt is $,
* Root user default prompt is #

**1978**

The **C shell(csh)** was developed by Bill Joy with the objective of achieving a scripting language similar to C programming language. This was useful given that C was a primary language in use back then which also made it easier and faster to use.

* Denoted as **csh**
* Non-root user default prompt is hostname %,
* Root user default prompt is hostname #.

**1983**

Developed by David Korn, the **Korn Shell(ksh)** combined features of both Bourne Shell and C Shell.

* denoted as **ksh**
* Non-root user default prompt is $**,**
* Root user default prompt is #

**1989**

One of the most widely used shells today, the **Bourne-Again Shell (bash)** was written by Brian Fox for the **GNU project** as a pre-software replacement for the Bourne Shell.

BASH evidently has more features than CSH and KSH since it has the features of all other shells in addition to its own. It is also more suitable for use by beginners, and learning it will introduce users to the other shells since their features are also being used by BASH.

# **How to determine Shell**

You can get the name of your shell prompt, with following command :

echo $SHELL or echo $0

**Installing different shells in Ubuntu**

sudo apt-get install csh

sudo apt-get install ksh

**You can find full path to your shell using the following command:**  
type -a bash  
type -a csh  
type -a ksh

### **To change your shell**

1. cat /etc/shells At the shell prompt, list the available shells on your system with cat /etc/shells.
2. chsh -s {shell-path-here}

chsh -s {shell-path-here} {user-name-here} Enter **chsh** (for "change shell"). Some systems prompt for a password, and some don't.

1. /bin/csh Type in the path and name of your new shell.

**ex:** chsh -s /bin/bash **ex:** chsh -s /bin/bash vivek

# How to Write Shell Script in Linux

**Shell Scripts** are written using text editors.

Let us understand the steps in creating a Shell Script:

1. **Create a file** **using** a **vi** editor(or any other editor). Name script file with **extension .sh**
2. Write some code.
3. Save the script file as filename.sh
4. For **executing** the script type **bash filename.sh**

## **What are Variables?**

A variable is a location in memory that is used to hold a value. This location is assigned a name to make it descriptive. The value could be any type of data such as a number or text. Therefore we can say that Variables store data in the form of characters and numbers.

Variables are defined as follows −

**variable\_name=variable\_value**

The following examples are valid variable names −

\_ALI

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invalid variable names −

2\_VAR

-VARIABLE

VAR1-VAR2

VAR\_A!

The reason you cannot use other characters such as **!, \*, or -** is that these characters have a special meaning for the shell.

## Variable Types

When a shell is running, three main types of variables are present −

* **Local Variables** − A local variable is a variable that is present within the current instance of the shell. Also called Shell Variables. A variable declared as *local* is one that is visible only within the block of code in which it appears. It has local "scope". In a function, a *local variable* has meaning only within that function block.
* **Environment Variables** − Environment variables contain information about your login session, stored for the system shell to use when executing commands. They exist whether you’re using Linux, Mac, or Windows. Many of these variables are set by default during installation or user creation . Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

For example, on Linux you can see your HOME environment variable's contents like this:

**$**echo**$HOME**  
**HOME=**/**home**/**ritesh** (note:home directory named *ritesh*)

To creat Environmental variable just type

export variable\_name=value

You can view all environment variables set on your system with the **env** command

* **Shell Variables** − An environment variable is available system wide and can be used by other applications on the system. A shell variable is a variable that is available only to the current shell. A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables. Shell variables are variables that apply only to the current shell instance. Each shell such as c**sh** and **bash**, has its own set of internal shell variables.

A shell variable is created with the following syntax:

"variable\_name=variable\_value".

# **Using Shell Arrays**

A shell variable is capable enough to hold a single value. These variables are called scalar variables.

Shell also supports a different type of variable called an **array variable**. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

Syntax: array\_name=(value1 ... valuen)

ex: msme=(Linux cloud data web)

## Accessing Array Values:

After you have set any array variable, you access it as follows −

To access the value stored in a variable, prefix variable name with the dollar sign (**$**)

${array\_name[index]}

You can access all the items in an array in one of the following ways −

${array\_name[\*]}

${array\_name[@]}

**Example:** Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

NAME01="Chakri"

NAME02="Teja"

NAME03="Habeeb"

NAME04="Sravanthi"

NAME05="Sandeep"

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable.

NAME[index]=value or NAME=(value1 ... valuen)

After you have set any array variable, you access it as follows −

${NAME[index]}

## **Shell Prompt**

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

**Printing or displaying output**

**echo** is one of the most commonly and widely used built-in command for Linux bash and C shells, that typically used in scripting language and batch files to display a line of text/string on standard output or a file

**Reading input data**

The **read** command takes the input from the keyboard and assigns it as the value to the variable PERSON

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

OUTPUT

Here is a sample run of the script −

What is your name?

Ritesh

Hello, Ritesh

$

**Shell Decision Making**

While writing a shell script, there may be a situation when you need to adopt one path out of the given paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Linux Shell supports conditional statements which are used to perform different actions based on different conditions. The two decision-making statements here −

1. The if...else statement
2. The case...esac statement

## **The if...else statements**

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Linux Shell supports following forms of **if…else** statement −

1. if...fi statement
2. if...else...fi statement
3. if...elif...else...fi statement

* **if...fi statement**

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally. The if command will executed only when the condition written in the **if** statement is ture.

## Syntax

if [ Condition ]

then

code to be executed if condition is true

fi

* **if...else...fi statement**

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

## Syntax

if [ Condition ]

then

code to be executed if condition is true

else

code to be executed if condition is not true

fi

* **if...elif...else...fi statement**

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

## Syntax

if [ Condition 1 ]

then

code to be executed if condition 1 is true

elif [ Condition 2 ]

then

code to be executed if condition 2 is true

elif [ Condition 3 ]

then

code to be executed if condition 3 is true

else

code to be executed if no condition is true

fi

**The case...esac statement**

Linux Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

The basic syntax of the **case...esac** statement is to give an Condition to evaluate and to execute several different statements based on the value of the Condition.

The interpreter checks each case against the value of the Condition until a match is found. If nothing matches, a default condition will be used.

## Syntax

case word in

pattern1)

code to be executed if pattern1 matches

;;

pattern2)

code to be executed if pattern2 matches

;;

pattern3)

code to be executed if pattern3 matches

;;

,

,

,

\*)

Default condition to be executed

;;

esac

**Linux - Shell Loop Types**

## To automatically perform a set of actions on multiple times we use loop command in shell programming.

The following types of loops available to shell programmers −

* The while loop
* The for loop
* The until loop

**The while loop**

The while loop enables you to execute a set of commands repeatedly until some condition occurs or the condition is true. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

while command

do

code to be executed if condition is true

done

**The for loop**

The **for** loop iterates over a list of items and performs the given set of commands. Each time the for loop executes, the value of the variable **var** is set to the next word in the list.

Syntax

for var in [LIST]

do

code to be executed

done

|  |  |  |
| --- | --- | --- |
| **Comparison** | **for loop** | **while loop** |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. |

**Until command**

Sometimes you need to execute a set of commands until a condition is False. Here, if the COMMANDS get evaluated to false then the statements will be executed. If the COMMANDS get evaluated to true then the no statements will be executed and control will go after the done statement.

## **Syntax**

until condition

do

Statement(s) to be executed until command is true

done

**Nesting Loops**

All the loops support nesting concept which means you can put one loop inside another similar one or different loops.

Syntax

while command1 ; # this is loop1, the outer loop

do

Statement(s) to be executed if command1 is true

while command2 ; # this is loop2, the inner loop

do

Statement(s) to be executed if command2 is true

done

Statement(s) to be executed if command1 is true

done

break and continue Statements

**break Statement**

The **break** statement terminates the current loop and passes program control to the command that follows the terminated loop. It is usually used to terminate the loop when a certain condition is met.

i=0

while [ $i -lt 5 ]

do

echo "Number: $i"

((i++)) #i=i+1

if [[ "$i" == '2' ]]; then

break

fi

done

echo 'All Done!'

**OUTPUT**

Number: 0

Number: 1

All Done!

**Continue Statement**

The continue statement exits the current iteration of a loop and passes program control to the next iteration of the loop.

i=0

while [ $i -lt 5 ]

do

((i++))

if [[ "$i" == '2' ]]; then

continue

fi

echo "Number: $i"

done

echo 'All Done!'

**OUTPUT**

Number: 1

Number: 3

Number: 4

Number: 5

All Done!

**Shell Basic Operators**

1. Arithmetic Operators
2. Relational Operators
3. Boolean Operators
4. String Operators
5. File Test Operators

**Note:** Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either awk or expr.

Example: `expr 2 + 2`

The complete expression should be enclosed between ` `, called the backtick.



It is not a single quote symbol.

## **Arithmetic Operators**

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \\* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| **Comparator operator** | | |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

## **Relational Operators**

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| **-eq** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a -eq $b ] is not true. |
| **-ne** | Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. | [ $a -ne $b ] is true. |
| **-gt** | Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true. | [ $a -gt $b ] is not true. |
| **-lt** | Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true. | [ $a -lt $b ] is true. |
| **-ge** | Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -ge $b ] is not true. |
| **-le** | Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true. | [ $a -le $b ] is true. |

**Boolean Operators**

The following Boolean operators also called as logical operator are supported by the Bourne Shell.

Boolean logic is a very easy way to figure out the truth of an Condition using the simple concept of true or false. In a nutshell, Boolean logic means you are working with stuff that is either TRUE or FALSE

Assume variable **a** holds 10 and variable **b** holds 20 then −

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| **!** | This is logical negation. This inverts a true condition into false and vice versa. | [ ! false ] is true. |
| **-o** | This is logical **OR**. If one of the operands is true, then the condition becomes true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| **-a** | This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

## **String Operators**

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| **=** | Checks if the value of two operands are equal or not; if yes, then the condition becomes true. | [ $a = $b ] is not true. |
| **!=** | Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true. | [ $a != $b ] is true. |
| **-z** | Checks if the given string operand size is zero; if it is zero length, then it returns true. | [ -z $a ] is not true. |
| **-n** | Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true. | [ -n $a ] is not false. |
| **str** | Checks if **str** is not the empty string; if it is empty, then it returns false. | [ $a ] is not false. |

## **File Test Operators**

We have a few operators that can be used to test various properties associated with a Linux file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**,**write** and **execute** permission on −

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example |
| **-r file** | Checks if file is readable; if yes, then the condition becomes true. | [ -r $file ] is true. |
| **-w file** | Checks if file is writable; if yes, then the condition becomes true. | [ -w $file ] is true. |
| **-x file** | Checks if file is executable; if yes, then the condition becomes true. | [ -x $file ] is true. |
| **-s file** | Checks if file has size **greater than** 0; if yes, then condition becomes true. | [ -s $file ] is true. |
| **-e file** | Checks if file **exists**; is true even if file is a directory but exists. | [ -e $file ] is true. |
| **-b file** | True if the file exists and is a block special file such as a hard drive like /dev/sda or /dev/sda1 | [ -b $file ] is false. |
| **-c file** | Checks if file is a character special file; if yes, then the condition becomes true.  **Character** **special** **files** or **character devices** provide unbuffered, direct access to the hardware **device**. They do not necessarily allow programs to read or write single characters at a time. | [ -c $file ] is false. |
| **-d file** | Checks if file is a directory; if yes, then the condition becomes true. | [ -d $file ] is not true. |
| **-f file** | Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.  **Ordinary files** contain ASCII (human-readable) text, executable program binaries, program data, and more | [ -f $file ] is true. |
| **-g file** | Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. | [ -g $file ] is false. |
| **-k file** | Checks if file has its sticky bit set; if yes, then the condition becomes true.  A Sticky bit is a permission bit that is set on a file or a directory that lets only the owner of the file/directory or the root user to delete or rename the file. | [ -k $file ] is false. |
| **-u file** | Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. | [ -u $file ] is false. |