A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

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.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

$date

Thu Jun 25 08:30:19 MST 2009

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells −

* **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
* **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

* Bourne shell (sh)
* Korn shell (ksh)
* Bourne Again shell (bash)
* POSIX shell (sh)

The different C-type shells follow −

* C shell (csh)
* TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

We are going to write many scripts in the next sections. It would be a simple text file in which we would put all our commands and several other required constructs that tell the shell environment what to do and when to do it.

Shell scripts and functions are both interpreted. This means they are not compiled.

Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example −

#!/bin/sh

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the****#****symbol is called a hash, and the ! symbol is called a bang*.

To create a script containing these commands, you put the shebang line first and then add the commands −

#!/bin/bash

pwd

ls

Shell Comments

You can put your comments in your script as follows −

#!/bin/bash

#Hi This is Preeti

pwd

ls

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

#!/bin/bash

#Hi This is Preeti

echo "What is your name?"

read PERSON

echo "Hello, $PERSON"

Here is a sample run of the script −

$./test.sh

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 [ expression ]

then

Statement(s) to be executed if expression is true

else

Statement(s) to be executed if expression is not true

fi

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows −

#!/bin/sh

a=10

b=20

if [ $a == $b ]

then

echo "a is equal to b"

else

echo "a is not equal to b"

fi

Upon execution, you will receive the following result −

a is not equal to b

**Introduction**

The bash **case** statement is the simplest form of the If. The **case** statement simplifies complex conditions with multiple different choices. This statement is easier to maintain and more readable than nested **if** statements.

The **case** statement tests the input value until it finds the corresponding pattern and executes the command linked to that input value. Thus, it is an excellent choice for creating menus where users select an option which triggers a corresponding action.

**In this tutorial, you will learn the bash case statement basics and how to use it in shell scripts.**

**Prerequisites**

* A machine running Linux.
* Access to the command line.
* A [text editor](https://phoenixnap.com/kb/best-linux-text-editors-for-coding) such as vi/vim.

**Bash case Statement Syntax**

The bash **case** statement takes the following syntax:

case $variable in

pattern-1)

commands;;

pattern-2)

commands;;

pattern-3)

commands;;

pattern-N)

commands;;

\*)

commands;;

esac

The **case** statement starts with the **case** keyword followed by the **$variable**and the**in**keyword. The statement ends with the **case** keyword backwards - **esac**.

**$variable**

* The script compares the input **$variable** against the patterns in each clause until it finds a match.

**Patterns**

* A pattern and its commands make a **clause**, which ends with **;;**.
* Patterns support special characters.
* The **)** operator terminates a pattern list.
* The **|** operator separates multiple patterns.
* The script executes the commands corresponding to the first pattern matching the input **$variable**.
* The asterisk **\*** symbol defines the default case, usually in the final pattern.

**Exit Status**

The script has two exit statuses:

* **0**. The return status when the input matches no pattern.
* **Executed command status**. If the command matches the input [variable](https://phoenixnap.com/kb/bash-function) to a pattern, the executed command exit status is returned.

**Bash case Statement Examples**

This section shows practical examples of using the bash **case** statement.

Example 1: Output a Description for Each Option

The following example is a script that lets the user choose a color and shows a comment corresponding to the input using the [echo command](https://phoenixnap.com/kb/echo-command-linux).

Follow the instructions below:

1. Open the terminal (**Ctrl + Alt + T**) and create the script:

vi color.sh

2. Add the following lines to the script:

#!/bin/bash

echo "Which color do you like best?"

echo "1 - Blue"

echo "2 - Red"

echo "3 - Yellow"

echo "4 - Green"

echo "5 - Orange"

read color;

case $color in

1) echo "Blue is a primary color.";;

2) echo "Red is a primary color.";;

3) echo "Yellow is a primary color.";;

4) echo "Green is a secondary color.";;

5) echo "Orange is a secondary color.";;

\*) echo "This color is not available. Please choose a different one.";;

esac

Each of the lines has the following role:

* The first line in each script is typically the shebang (**#!**), which instructs the operating system which interpreter to use to parse the file.
* Lines 2-7 are the options menu presented to the user to choose from.
* The **read color;** variable prompts the user for an answer and stores it.
* The **case** statement contains the possible responses that correspond to the user's input. In this example, the output is the **echo** command output.
* The last line, **esac**, ends the **case** statement.

3. Save the script and [exit vi](https://phoenixnap.com/kb/how-to-exit-vim) with the following command:

:wq

4. Before running each script, make sure to chmod it to make it executable:

chmod +x color.sh

5. [Run the script](https://phoenixnap.com/kb/run-bash-script):

./color.sh



The script offers the options menu and outputs a different message depending on the selected option.

Example 2: Using Multiple Patterns

The **case** statement allows using multiple patterns in each clause. If the expression matches the specified patterns, the script executes the commands in that clause.

This example script prompts the user to enter a month and outputs the number of days. There are three possible answers:

* 30 days.
* 31 days.
* 28 or 29 days for February.

Follow the steps below to create the script:

1. Create the script *month.sh* by running:

vi month.sh

2. Enter the following lines and save the script:

#!/bin/bash

shopt -s nocasematch

echo "Enter the name of a month."

read month

case $month in

February)

echo "There are 28/29 days in $month.";;

April | June | September | November)

echo "There are 30 days in $month.";;

January | March | May | July | August | October | December)

echo "There are 31 days in $month.";;

\*)

echo "Unknown month. Please check if you entered the correct month name: $month";;

esac

In the example above:

* The **shopt** command with the **-s nocasematch** option reduces the chance for error with the pattern matching being case insensitive.
* The pipe symbol **|** separates the patterns in each of the clauses.

3. Make the script executable:

chmod +x month.sh

4. Run the script:

./month.sh



The script ignores differences in case and outputs the correct number of days in the selected month.

**Shell Scripting for loop**

The for loop moves through a specified list of values until the list is exhausted.

**1) Syntax:**

Syntax of for loop using **in** and list of values is shown below. This for loop contains a number of variables in the list and will execute for each item in the list. For example, if there are 10 variables in the list, then loop will execute ten times and value will be stored in varname.



* Keywords are for, in, do, done
* List is a list of variables which are separated by spaces. If list is not mentioned in the for statement, then it takes the positional parameter value that were passed into the shell.
* Varname is any variable assumed by the user.

Example for:

We have shown an example to count 2's table within for loop.



Look at the above snapshot, our varname is **table**, list is specified under curly braces. Within the curly braces, first two will initialize the table from 2, 20 represents maximum value of $table and last 2 shows the increment by value 2.



Look at the above snapshot, it displays the 2's table as the output.

**2) Syntax:**

Syntax of for like C programming language.



Look at the above snapshot, condition1 indicates **initialization**, cond2 indicates **condition** and cond3 indicates **updation.**

Example for:

We have shown an example to count the number in reverse direction.



Look at the above snapshot, this is the loop script. $i will initialize with 10 and will go till 1, decrementing with 1 value.



Look at the above snapshot, this is the output of the script.

**Shell Scripting while loop**

Linux scripting while loop is similar to C language while loop. There is a condition in while. And commands are executed till the condition is valid. Once condition becomes false, loop terminates.

**Syntax:**

Syntax of while loop is shown in the snapshot below,



Example:

We have shown the example of



Output is displayed in the below snapshot,



while infinite loop:

Infinite loop is also called endless loop. It is made with **while true** (it means condition will always be true) or **while** : (it means an empty expression), where colon (:) is equivalent to no operation.



Look at the above snapshot, this script includes **while true**syntax.



Look at the above snapshot, this script includes **while:** syntax.

Both of them display the same output.

**Use nc command to check the remote port is open in Linux**

$ nc [-options] [HostName or IP] [PortNumber]

$ nc -zvw10 192.168.0.1 22

* z: zero-I/O mode which is used for scanning
* v: for verbose output
* w10: timeout wait 10 seconds

The “nc” command stands for “netcat”. The “nc” command is a very versatile command that can be used for a variety of purposes, including network administration and data transmission.

For example, the “nc” command can be used to create a simple [TCP connection](https://www.howtouselinux.com/post/tcp-connection) between two computers.  The “nc” command can be used to connect to a remote server on a given port and send/receive data.

For example, if you want to connect to a remote server on port xx, you would use the following command: nc  -zv <remote server> port

In this example, “<remote server>” is the IP address or hostname of the remote server, and “<port>” is the port that you want to connect to.

I needed to see if the port 22 (SSH) on a remote machine was open, so I opened a terminal and ran the following command:

$ nc -vz hostname.com 22

The -v option enabled verbose output, and the -z option instructed nc to only scan for open ports, without actually establishing a connection.

The output showed me the results of the port scan:

Connection to hostname.com 22 port [tcp/ssh] succeeded!

This told me that the port 22 was open and that I could connect to the remote machine using SSH.

In another scenario, if the port was not open, the output would look something like this:

nc: connect to hostname.com port 22 (tcp) failed: Connection refused

You can also use the “nc” command to [open a port in Linux](https://www.howtouselinux.com/post/linux-command-open-a-port-on-linux). To do this, you would use the following command: nc -l -p 1234

In this example, “-l” is used to listen for a connection on port 1234

***history***command is used to view the previously executed command. This feature was not available in the Bourne shell. Bash and Korn support this feature in which every command executed is treated as the event and is associated with an event number using which they can be recalled and changed if required. These commands are saved in a history file. In Bash shell **history** command shows the whole list of the command.

**Syntax:**

$ history



Here, the number(termed as event number) preceded before each command depends on the system. You may get different numbers while executing on your own system.

History in shell script

* To show the limited number of commands that executed previously as follows:

$ history 5



**Note:** The command can be executed using event number also.

*Example:*

$ !1997





* To print command before executing so that a wrong command does not get executed use **:p** after event number of command.

**Example:**

!1997:p



* This command can also be used along with grep.

**Example:**

history | grep chpasswd



* The most recent command can be viewed using **!!**

**Example:**

!!



* Suppose if the command has to be executed without being stored variable **HISTFILE** has to be unset.

**Example:**



* The command can also be executed using a part of the command.

**Example:**

!command\_starting\_string



* History can also be removed using **history -d event\_number**

**Example:**

history -d 1996



* The whole history can be removed using **history -c** option.

**Example:**

history -c



history | tail

* It is used to view the last 10 commands

