Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Linux - Introduction to shell/60cc9fe4cfaf5f6628f2d42b

TUTORIAL

Linux - Introduction to shell

Topics

Introduction

The shell is a command interpreter. More than just the insulating layer between the operating system kernel and the user, it's also a fairly powerful programming language. A shell program, called a script, is an easy-to-use tool for building applications by "gluing together" system calls, tools, utilities, and compiled binaries. Shell scripts are especially well suited for administrative system tasks and other routine repetitive tasks not requiring the bells and whistles of a full-blown tightly structured programming language.

A working knowledge of shell scripting is essential to anyone wishing to become reasonably proficient at system administration, even if they do not anticipate ever having to actually write a script. Consider that as a Linux machine boots up, it executes the shell scripts in /etc/rc.d to restore the system configuration and set up services. A detailed understanding of these startup scripts is important for analyzing the behavior of a system, and possibly modifying it. The craft of scripting is not hard to master, since scripts can be built in bite-sized sections and there is only a fairly small set of shell-specific operators and options to learn.

Suppose you want to look up a filename, check if the associated file exists, and then respond accordingly, displaying a message confirming or not confirming the file's existence. If you only need to do it once, you can just type a sequence of commands at a terminal.

However, if you need to do this multiple times, automation is the way to go. In order to automate sets of commands, you will need to learn how to write shell scripts, the most common of which are used with bash. The main features of shell script are as below:

- Automate tasks and reduce risk of errors
- Combine long and repetitive sequence of commands into one simple command
- Share procedures among several users
- Quick prototyping and no need of compiling
- Create new commands using combination of utilities
- Provide a controlled interface to users

Commands shell examples

The command interpreter is tasked with executing statements that follow it in the script. Commonly used interpreters include: **/usr/bin/perl, /bin/bash, /bin/csh, /usr/bin/python and /bin/sh**.

Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

- /bin/sh
- /bin/bash
- /bin/tcsh
- /bin/csh
- /bin/ksh
- /bin/zsh

Most Linux users use the default bash shell, but those with long UNIX backgrounds with other shells may want to override the default. We will be using Bash, an acronym for "Bourne-Again shell" and a pun on Stephen Bourne's now classic Bourne shell. Bash has

become a de facto standard for shell scripting on most flavors of Linux/Unix.

Environment Variables

Every active shell stores pieces of information that it needs to use in what are called environment variables. An environment variable can store things such as locations of configuration files, mailboxes, and path directories. They can also store values for your shell prompts, the size of your history list, and type of operating system.

Most scripts use variables containing a value, which can be used anywhere in the script. These variables can either be user or system-defined. Many applications use such environment variables for supplying inputs, validation, and controlling behavior.

Some examples of standard environment variables are HOME, PATH, and HOST. When referenced, environment variables must be prefixed with the $ symbol, as in $HOME. You can view and set the value of environment variables. For example, the following command displays the value stored in the PATH variable:

To see the environment variables currently assigned to your shell, type the declare command. (It will probably fill more than one screen, so type declare | more. The declare command also shows functions as well as environment variables.) You can refer to the value of any of those variables by preceding it with a dollar sign ($) and placing it anywhere on a command line. For example:

```
$ echo $USER
user1
```

This command prints the value of the USER variable, which holds your username (user1). Substitute any other value for USER to print its value instead.

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the MYCOLOR variable to blue:

```
$ MYCOLOR=blue
```

You can get a list of environment variables with the env, set, or printenv commands.

HOME and PATH are two mostly used environment variables. Environment variables can provide a handy way to store bits of information that you use often from the shell. You can create any variables that you want by avoiding those that are already in use. To set an environment variable temporarily, you can simply type a variable name and assign it to a value. Here's an example:

```
$ FAVDIR=/home/user1/project1/oct17/codequotient
$ export FAVDIR
```

This example causes a long directory path to be assigned to the FAVDIR variable. The export FAVDIR command says to export the value to the shell so that it can be propagated to other shells you may open. With FAVDIR set, you go to the directory by typing the following:

```
$ cd $FAVDIR
```

The problem with setting environment variables in this way is that as soon as you exit the shell in which you set the variable, the setting is lost. To set variables permanently, add variable settings to a bash configuration file.

Exporting Environment Variables

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the export statement, as in:

```
$ export VAR=value
```

or

```
$ VAR=value
$ export VAR
```

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, they are only copied and inherited.

Typing export with no arguments will give a list of all currently exported environment variables.

CodeQuotient

Tutorial by codequotient.com | All rights reserved, CodeQuotient 2025

codequotient.com